Visual Studio vs. RAD Studio
Comparativo entre os IDEs, para desenvolvedores Delphi Prism – Parte 2

Clube Delphi

Firebird 2.5
Trace API

XML Data Binding
Manipule arquivos XML como objetos

Ano 9 :: Edição 128 :: R$ 11,90

DevMedia

Integração entre as plataformas Win32 e .NET

Brinde

Comprando esta edição ganhe 4 vídeo-aulas:

- Delphi Prism - Web – Profiles
- Backup do MySQL via Delphi

DataSnap
Aplicações Desktop e Web acessando o mesmo AppServer

Delphi Prism
Conheça os melhores recursos da linguagem – Parte 2

ISSN 1517990-7

Dificuldades em encontrar conteúdo sobre .Net para iniciantes?

A DevMedia simplifica pra você!

easy .net magazine

Easy .Net Magazine:

A primeira revista digital feita especialmente para quem está dando os primeiros passos em .Net!

Você encontrará exemplos e modelos fáceis e práticos, noções introdutórias e conceitos básicos.

Conheça as vantagens:

- Vídeos e apresentações "embutidos" nas páginas;
- Ilustrações, diagramas e modelos apresentados em PowerPoint;
- Entrevistas em vídeo;
- Códigos-fonte que podem ser copiados com CTRL+C/CTRL+V.

A Easy.Net não é feita em PDF ou Flash!
Ela utiliza um aplicativo 100% online.

DevMedia group

Acesse agora a edição Nº 1 totalmente grátis e comprove

www.devmedia.com.br/easynet

Easy.Net Magazine, seu curso inicial de .Net!

Sumário

Quick Update

06 – A evolução das tecnologias multicamadas
[*Daniel Sobrinho Laporte*]

Delphi

Expert

08 – Interoperabilidade
Integre projetos .NET e Win32
[*Vitor Luiz Rubio*]

Banco de Dados

17 – Trace API do Firebird 2.5
Faça a auditoria do que acontece com o banco de dados
[*Jederson Zuchi*]

Novidades – Mini-Curso

24 – Delphi Prism
Melhorias da linguagem – Parte 2
[*Rodrigo Araujo*]

Boas Práticas

Boas Práticas

30 – DataSnap no Delphi XE
Conectando um mesmo servidor de aplicação a partir de clientes Web e Desktop
[*Fabrício Hissao Kawata*]

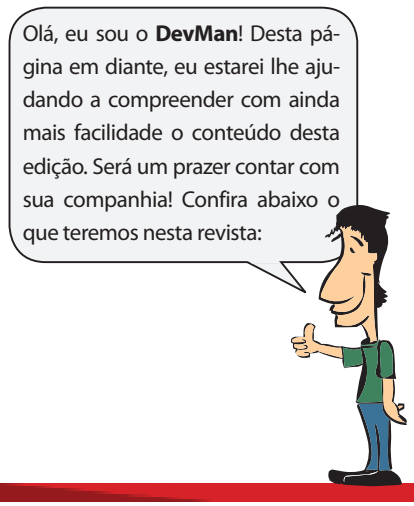
Boas Práticas

38 – XML Data Binding
Manipule arquivos XML de forma fácil e rápida
[*Jéfferson Ricardo Zuchi*]

Easy Delphi

Easy Delphi – Mini-Curso

44 – Visual Studio vs. RAD Studio
Comparativo entre os IDEs, para desenvolvedores Delphi Prism – Parte 2
[*Rodrigo Araujo*]



Brinde na web desta edição - 2 vídeos! Confira:

2 Vídeos

- **Delphi Prism – Introdução a profiles – Desenvolvimento web**
  http://www.devmedia.com.br/post-20036-Introducao-a-Profiles-Desenvolvimento-Web-Curso-Completo-Delphi-Prism-Parte-82.html

- **Backup MySQL – Rotina de backup do MySQL via Delphi**
  http://www.devmedia.com.br/post-19123-Backup-MySQL-Curso-Aplicacao-Financeira-Delphi-2009-e-MySQL-Parte-50.html

Para acessar os vídeos utilize os seguintes dados
Login: **DVM.PLS** *Senha:* **ZNN90M**

Gostou das vídeo aulas? O portal *www.devmedia.com.br* possui mais de **2 mil vídeo aulas** e **dezenas de cursos online** sobre desenvolvimento de software! Agora você pode comprar as vídeo aulas que preferir e fazer sua própria combinação de vídeos! Saiba mais em *www.devmedia.com.br/creditos*

Dê seu feedback sobre esta edição!

A ClubeDelphi tem que ser feita ao seu gosto. Para isso, precisamos saber o que você, leitor, acha da revista!

Dê seu voto sobre esta edição, artigo por artigo, através do link:

www.devmedia.com.br/clubedelphi/feedback

Para votar, você vai precisar do código e senha de banca desta edição, que é:
Login: **DVM.PLS** Senha: **ZNN90M**

Clube Delphi

Ano 9 - 128ª Edição 2011 - ISSN 1517990-7 Impresso no Brasil

Corpo Editorial

Editor Geral
Guinther Pauli
guinther@devmedia.com.br

Editor Técnico
Paulo Quicoli
pauloquicoli@devmedia.com.br

Daniel Laporte
daniel.sobrinho@devmedia.com.br

Equipe Editorial
Vitor Luiz Rubio
vitorrubio@gmail.com

Rafael Stavarengo
faelsta@gmail.com

Jornalista Responsável
Kaline Dolabella - JP24185

Capa e Diagramação
Romulo Araujo - romulo@devmedia.com.br

Revisão
Caroline Velozo - carolinelopes@devmedia.com.br

Coordenação Geral
Daniella Costa - daniella@devmedia.com.br

Distribuição
Fernando Chinaglia Dist. S/A
Rua Teodoro da Silva, 907
Grajaú - RJ - 206563-900

Atendimento ao Leitor
A DevMedia conta com um departamento exclusivo para o atendimento ao leitor. Se você tiver algum problema no recebimento do seu exemplar ou precisar de algum esclarecimento sobre assinaturas, exemplares anteriores, endereço de bancas de jornal, entre outros, entre em contato com:

Isabelle Macedo e Uellen Goulart – Atendimento ao Leitor
www.devmedia.com.br/mancad
(21) 3382-5038

Kaline Dolabella – Gerente de Marketing e Atendimento
kalined@terra.com.br
(21) 3382-5038

Publicidade
Para informações sobre veiculação de anúncio na revista ou no site e para fechar parcerias ou ações específicas de marketing com a DevMedia, entre em contato com:

Cristiany Queiróz

publicidade@devmedia.com.br

EDITORIAL

A migração já pode ser considerada uma fase natural do ciclo de vida de um software. Seja migrar de uma arquitetura para outra, por exemplo, de client/server para multicamadas, ou de plataforma, Win32 para .NET. Sistemas Desktop passam para a Web / nuvem, ou para Smartphones. Migrações mais radicais podem significar também o abandono completo da linguagem e ferramenta de desenvolvimento, por exemplo, a adoção do Visual Studio e C#. Independentemente da migração, sair portando toda a aplicação de uma única vez pode se tornar um desafio muito grande para a equipe de desenvolvimento, o que envolve muitos riscos. É necessário conhecimento técnico na plataforma destino, são necessários prazos, existem custos, e o risco de fracasso deve ser analisado. A arquitetura de um projeto pode fornecer preciosas pistas de como o software pode ser migrado. Uma aplicação multicamadas pode ser migrada no estilo horizontal. Uma camada servidora toda pode ser portada de Win32 para .NET, por exemplo, de DataSnap para WCF. As aplicações clientes através de padrões de interoperabilidade, como XML, continuam a funcionar sem alterações, até que também sejam migradas (por exemplo de VCL para Silverlight). Outro tipo de migração é a vertical, ideal para soluções modularizadas. Migra-se o servidor e cliente, porém apenas de uma determinada parte da solução. O conhecimento adquirido nesta etapa pode servir para os outros módulos. É importante analisar as vantagens e desvantagens de cada estratégia, e escolher aquela que melhor se adapta às necessidades da empresa. É exatamente nesse ponto que entra em cena um importante recurso da programação: a Interoperabilidade. Por anos a plataforma Microsoft procurou oferecer os mais diversos tipos e tecnologias para promover o diálogo entre diferentes plataformas: DLLs "puras", DDE, COM, COM+ etc. Tecnologias mais modernas baseadas no .NET facilitam muito a vida do desenvolvedor, principalmente porque tudo é baseado nas especificações CLS e CTS. Em quase todos os cenários, interligar Win32 e .NET se tornou uma necessidade, seja em estratégias de migração horizontal ou mesmo para interligar diferentes aplicações de diferentes fabricantes / empresas. Este é o tema de destaque desta edição da ClubeDelphi, onde o Vitor Rubio mostrará como promover a comunicação entre as plataformas Win32 e .NET usando diferentes técnicas de programação.

Ainda nesta edição: confira na matéria do Jederson Zuchi um especial sobre a nova API de Trace do Firebird 2.5, permitindo que o desenvolvedor / DBA monitore o que ocorre no BD. O Rodrigo Araujo traz dois artigos para os desenvolvedores Delphi Prism. Já domina o Delphi Win32? Veja que outros interessantes recursos temos no compilador do Prism, e conheça o Visual Studio comparando ao que já conhece no RAD Studio. O Fabricio Kawata traz uma excelente solução nesta edição: veja como é simples criar uma solução de arquitetura baseada em DataSnap que tem diferentes interfaces clientes, por exemplo, uma Web e outra Desktop, compartilhando o mesmo acesso a dados e regras de negócio. Confira também como é simples trabalhar com documentos XML se forem tratados como objetos, usando XML Data Bindings, na matéria do Jéfferson Zuchi.

Guinther Pauli
guinther@devmedia.com.br
Certificado Delphi 3,5,6,7,2005,2006,Web,Linux
Microsoft Certified: MCP, MCAD, MCSD.NET, MCTS, MCPD (ASP.NET,C#)
guintherpauli.blogspot.com - twitter.com/guintherpauli

Fale com o Editor!

É muito importante para a equipe saber o que você está achando da revista: que tipo de artigo você gostaria de ler, que artigo você mais gostou e qual artigo você menos gostou. Fique a vontade para entrar em contato com os editores e dar a sua sugestão!
Se você estiver interessado em publicar um artigo na revista ou no site ClubeDelphi, entre em contato com os editores, informando o título e mini-resumo do tema que você gostaria de publicar:

Guinther Pauli - Editor da Revista
guinther@devmedia.com.br

ClubeDelphi PLUS

A revista ClubeDelphi é parte integrante da assinatura ClubeDelphi PLUS.
Para mais informações sobre o pacote PLUS, acesse:
http://www.devmedia.com.br/clubedelphi/portal.asp

DevMedia

Sistema de créditos DevMedia

Conteúdo **completo** ao seu alcance

Com o sistema de créditos DevMedia você pode comprar individualmente mais de 4.000 vídeo aulas, 8000 artigos, além de palestras e vídeos com formações completas.

TODAS as vídeo aulas que você sempre quis ver a partir de

R$ 0,99

ECONOMIA: você não precisa assinar 8 produtos diferentes para ter acesso ao conteúdo completo do site!

Saiba mais sobre o Sistema de Créditos!

www.devmedia.com.br/creditos

DevMedia

A evolução das tecnol

Com a chegada da versão 2009 do Delphi a arquitetura multicamadas entrou em ascensão. Nunca os desenvolvedores tiveram tamanho interesse em desenvolvimento de aplicações distribuídas e assim, o modelo Cliente-Servidor segue deixando de ser a principal opção para empresas ou desenvolvedores independentes que têm iniciado algum projeto.

O modelo multicamadas proporciona inúmeras vantagens em relação à arquitetura cliente-servidor, dentre estas se pode citar a distribuição geográfica, que permite que o servidor de aplicação e a aplicação cliente sejam executadas de localidades diferentes e com alta-performance (superando ou igualando até mesmo o modelo cliente-servidor, dependendo da forma em que ambos foram desenvolvidos), facilidade de manutenção, pois permite que apenas a aplicação cliente seja disponibilizada, reduzindo os problemas do processo de atualização entre tantas outras vantagens. Mas talvez a principal vantagem deste crescimento esteja na evolução desta tecnologia, que é o que aborda este Quick Update.

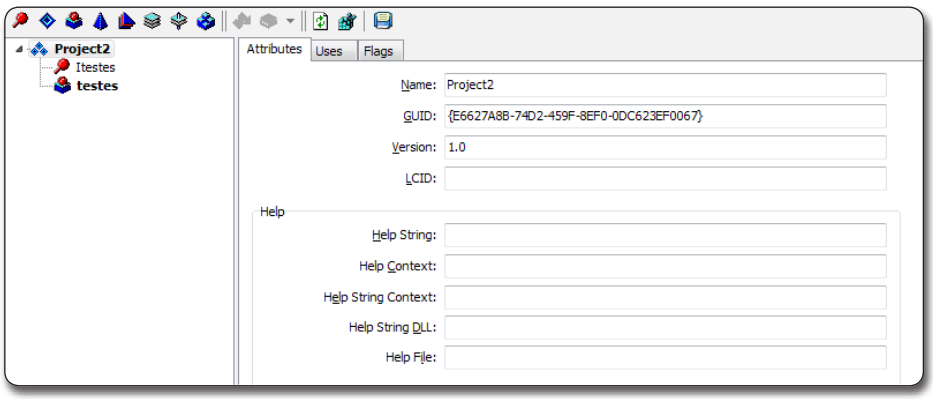


Figura 1. Type Library

Antes da chegada da nova arquitetura de multicamadas baseada em DataSnap (referindo-se ao modelo atual do Delphi XE), houve várias tecnologias baseadas neste conceito de aplicação distribuídas como DCOM, COM+ e BSS, e é claro que todas estas tecnologias como parte de um quadro evolutivo, apresentando inovações, melhorias, mas com dezenas de problemas que foram sendo aperfeiçoados até chegar no modelo atual.

Dentre estas pode-se destacar o DCOM (Distributed Component Object Model) e COM (Component Object Model). Estas eram tecnologias que tiveram início nas versões 4 e 5 do Delphi e tinham seus servidores baseados em DLLs. O principal problema destas tecnologias era a queda de performance do servidor com o crescimento de conexões clientes, o que mudou com o COM+.

A tecnologia COM+ foi introduzida pela Microsoft e fez com que várias tecnologias se baseassem nela para seus surgimentos. A tecnologia COM+ ainda se caracterizava por compor o servidor de aplicação em DLL e fazia o uso de GUIDs (Globally Unique Identifier) para registrar os Transactional Data Modules, que eram basicamente Data Modules que implementavam a Interface IAppServer.

Uma das tecnologias derivadas do COM+ foi o BSS (Borland Socket Server). Todo este canal de comunicação entre as camadas é realizado pelo BSS (Borland Socket Server), que nada mais é do que uma tecnologia baseada em DCOM, funcionando por meio de Sockets que diferente de SOAP que se comunicam através de XML, os Sockets comunicam-se por meio de protocolos TCP ou UDP.

O BSS ao contrário do COM+ utiliza-se de um servidor de aplicação baseado em executável, e não uma DLL como o servidor utilizado pelo COM+. Para seu funcionamento utilizava-se do Socket Server, que era como um túnel de comunicação entre aplicação cliente e servidora. Vale lembrar que o BSS possui uma boa performance, porém ainda assim possui a dependência da tecnologia COM.

Vale lembrar que ambas as tecnologias faziam uso da Type Library (**Figura 1**) utilizada para administrar os métodos e Remote/Transactional Data Modules. Essa que por sinal, é um dos grandes

Nota do DevMan

NRT: Apesar do COM+ ser compilado em forma de DLL, no servidor ele fica hospedado em um processo executável (host), justamente pelo fato de que DLLs não existem sozinhas na memória.

Nota do DevMan

Globally Unique Identifier ou GUIDs são identificadores gerados por cadeias de 32 caracteres. Estes GUIDs podem ser facilmente encontrados no registro do Windows com o nome do Remote Data Module ou Transactional Data Module. No Delphi é criada através do atalho Ctrl + Shift + G. Exemplo:

['{B6AD21BA-28DA-4AFC-A50D-52346D61F791}']

ogias multicamadas

Daniel Sobrinho Laporte

problemas destas tecnologias, possuindo diversos problemas durante a manutenção e criação dos métodos.

Estas tecnologias seguiram até a versão do Delphi 2009, onde teve início o novo modelo do DataSnap, baseando-se no DBX Framework. Com isto, o principal problema das tecnologias foi resolvido, a dependência do COM.

O novo DataSnap surgiu com maior desempenho que seu antecessor, trazendo novos componentes e uma maneira de desenvolvimento completamente diferenciada, utilizando por exemplo, recursos da classe Proxy gerada pelo DBX Framework. Algo que faltou nesta versão foi o suporte a Callback (apesar de ser suportado no COM+), que após ser requisitado pelos desenvolvedores da comunidade, teve sua volta na versão 2010 através do suporte à JSON.

Além de todas as melhorias descritas aqui e presentes na evolução da tecnologia multicamadas, pode-se considerar as outras novidades do Delphi que logicamente podem ser incluídas no desenvolvimento multicamadas e que estão presentes desde a versão 2009, algo como: melhoria na RTTI, suporte a Touch e Gestures, driver de conexão nativo para Firebird, Generics entre muitas outras coisas.

Com este overview é possível ver a escala de evolução não só da arquitetura multicamadas, mas também do Delphi, o que torna a comunidade cada vez mais confiante no sucesso e crescimento do Delphi.

Daniel Sobrinho Laporte
daniel.sobrinho@devmedia.com.br
daniel.laporte@gmail.com
www.twitter.com/daniel_laporte
Colunista e editor técnico da Revista ClubeDelphi. Graduado em Sistemas de informação e pós-graduado em Administração de Bancos de Dados pela Universidade de Franca. É analista de sistemas e desenvolvedor Delphi desde 2005 desenvolvendo aplicações cliente/servidor e multicamadas.

Blogs do autor
http://twitter.com/daniel_laporte

Dê seu feedback sobre esta edição!

A Clubedelphi tem que ser feita ao seu gosto. Para isso, precisamos saber o que você, leitor, acha da revista!

Dê seu voto sobre este artigo, através do link:

www.devmedia.com.br/clubedelphi/feedback

Faça um *upgrade* em sua carreira

Em um mercado cada vez mais focado em qualidade, ter conhecimentos aprofundados sobre requisitos, metodologia, análises, testes, entre ouutros, pode ser a diferença entre conquistar ou não uma boa posição profissional. Sabendo disso a DevMedia lançou uma publicação totalmente especializada em Engenharia de Software. Todos os meses você pode encontrar artigos sobre Metodologias Ágeis; Metodologias tradicionais (document driven); ALM (application lifecycle); SOA (aplicações orientadas a serviços); Análise de sistemas; Modelagem; Métricas; Orientação à Objetos; UML; testes e muito mais. **Assine Já!**

DevMedia

Interoperabilidade

Integre projetos .NET e Win32

Há várias situações em que é necessário integrar ambientes diferentes. As mais comuns são quando uma empresa adquire um conjunto de bibliotecas que não são feitas nativamente na linguagem onde ele deseja usar, por exemplo, compra uma coleção de DLLs feitas em C++ para usar no C#, ou quando a empresa contrata um terceiro apenas para desenvolver a biblioteca. Mais comum ainda são as integrações feitas via Web Services SOAP e via HTTP puro, podendo usar ou não REST e podendo usar ou não XML.

Um importante conceito nesses casos é determinar o que é o componente, entendê-lo como uma caixa preta, criar um programa simples para testá-lo e só então integrá-lo na aplicação principal. Do ponto de vista da aplicação principal esse componente será apenas um método, ou uma unit com um conjunto de métodos.

Para usar DLL é necessário que as funções desta sejam declaradas com stdcall e que se use PChar no lugar de strings. Também é possível usar uma biblioteca do .NET no Delphi. Para isso a biblioteca deve ter um strong name, deve estar no GAC e deve ter a propriedade ComVisible(true) nos parâmetros da biblioteca.

Também é possível a comunicação entre programas e componentes via Web Services, mas, ampliando um pouco os horizontes e pensando mais abstratamente: Web Service é qualquer "site" hospedado na web que fornece um serviço ou funcionalidade. Sendo assim um Web Service não precisa necessariamente seguir o protocolo SOAP, que é bastante extenso, mas pode se beneficiar de qualquer chamada HTTP pura que retorne um XML como resultado.

Resumo DevMan

De que trata o artigo:

Trata da habilidade de acessar componentes .NET a partir do Delphi, ou acessar componentes do Delphi a partir do .NET, acessar serviços web feitos em ambas as linguagens ou ainda trocar mensagens via API do Windows.

Para que serve:

Não é possível hoje em dia, com os prazos diminuindo e os requisitos aumentando, criar-se um sistema do zero. Sempre é necessário partir de algo pronto, de um legado, ou comprar e integrar bibliotecas e serviços de terceiros, não importando a tecnologia com que foram feitos.

Em que situação o tema é útil:

Em sistemas híbridos, multiplataforma, distribuídos e integrados é extremamente necessário que se reutilizem componentes de software já prontos. Este artigo será útil para todas as ocasiões em que for necessário acessar um Web Service feito em .NET pelo Delphi, acessar uma DLL feita em Delphi pelo .NET, trocar arquivos de texto ou XML entre as duas plataformas, monitorar mensagens na API do Windows ou até fazer com que aplicações se comuniquem diretamente via sockets, onde é necessário interoperabilidade.

Interoperabilidade:

Neste artigo são mostradas algumas técnicas de interoperabilidade entre aplicações Delphi Prism (e por extensão C# ou qualquer outra compatível com .NET Framework) e Delphi Win32 nativas. São mostrados métodos de se integrar os dois ambientes via DLLs, via mensagens da API do Windows, via Web Services e outros métodos mais "exóticos". O objetivo final do artigo é tentar ao máximo criar uma aplicação nova, em .NET, sobre uma aplicação Win32 nativa, aproveitando-se tudo que for possível e com mínimo esforço ou mínimo de alterações na aplicação legada.

Outro meio de se implementar a comunicação entre os sistemas é através de troca de mensagens pela API do Windows. Nesta forma o componente ou sistema legado em questão não é um conjunto de DLLs mas sim um programa rodando.

Um ponto muito importante para a integração entre sistemas é a serialização de objetos complexos e estruturas (structs ou records). É possível serializar objetos no formato XML, JSON, Yammel ou TXT. Algumas bibliotecas do Delphi ou do .NET podem auxiliar nesta tarefa, mas esses métodos de serialização podem ser implementados manualmente também. Nem todos os métodos de serialização serão abordados neste artigo.

Comunicação direta via sockets, usando protocolo próprio, pode ser interessante para se passar parâmetros ou objetos serializados entre aplicações. Abstratamente é isso que aplicações DataSnap e Web Services fazem. Em último caso pode-se recorrer ao método mais antigo de todos para integração entre sistemas: troca de arquivos TXT.

DLLs

Assim como o Visual C++ e qualquer outro compilador da Microsoft a direção

ou convenção de chamada de métodos é, por padrão, stdcall, herança do C/C++. No Delphi a convenção de chamada utilizada é a FastCall. Nada impediria de mudar isso, mas o que é padrão em um ambiente tem de ser avisado no outro. O que torna isso impossível é o fato de que a convenção FastCall não foi padronizada, tendo sido implementada no Delphi e no .NET/C++ de maneira diferente. Por isso FastCall é incompatível entre Delphi e .NET. Use sempre as convenções padronizadas, dando preferência sempre a stdcall, que é o mais universal.

É possível usar stdcall no Delphi, basta usar a diretiva stdcall depois do nome do método. Mandamentos para usar DLLs do Delphi no Prism:

1. Use apenas tipos primitivos;
2. Strings NÃO SÃO tipos primitivos;
3. Não use sharemem e borlandmm.dll porque esses mecanismos servem apenas para transferir strings do Delphi entre aplicações e DLLs feitas em Delphi. Além das strings do Delphi terem um formato ou implementação diferente das outras linguagens, a transferência de strings nativas depende da convenção de chamada FastCall e do gerenciador de memória do Delphi, os quais não serão usados neste cenário;
4. Use PChar e PAnsiChar no lado Delphi;
5. No lado .NET, strings são transformados pelo DLLImport em AnsiStrings. Para especificar WideStrings use CharSet.Unicode;

A **Listagem 1** mostra o código do lado Delphi e a **Listagem 2** do lado Prism. O objetivo dessas primeiras listagens foi mostrar como uma integração pode ser feita através de DLLs feitas em Delphi e utilizadas no Prism. A declaração de métodos estáticos com convenção de chamada stdcall é essencial para isso. É possível também ter forms criadas dentro dessas DLLs contanto que elas sejam criadas e destruídas dentro da DLL.

Na **Listagem 1** a cláusula Exports lista quais métodos de dentro da DLL serão visíveis para uma aplicação que importe e use essa DLL. Numa tradução livre Exports "exporta" mesmo os métodos da DLL. Na **Listagem 2** o atributo DLLImport usa os serviços de interoperabilidade do .NET e faz de fato a importação das funções da DLL que estão sendo exportadas. É o equivalente a "external nomedll.dll' name 'nomefuncao';" que se usa no Delphi Win32 depois da declaração de uma função para indicar que esta vem de uma DLL.

São mostradas chamadas a três métodos: um deles soma dois inteiros dentro da DLL e retorna o resultado para o Prism, o outro simplesmente recebe uma string como parâmetro e concatena com outra string dentro da DLL, retornando uma cadeia maior. Esta última função possui versões ANSI e WIDE.

Nota do DevMan

Convenções de chamada explicam para o sistema, em baixo nível, qual a ordem dos parâmetros de um método, quais serão trazidos dos registradores e quais serão trazidos da memória e "de quem" é a responsabilidade de limpar a pilha: de quem está chamando ou de quem é chamada. A diferença entre stdcall e fastcall é que enquanto stdcall foi padronizado por normas ISO/ANSI, fastcall pode ser implementado diferente de acordo com o fabricante do compilador ou da linguagem. O fastcall do Delphi Win32 não encontra correspondente nas outras linguagens. Para compreender as diferenças entre as convenções de chamada é necessário compreender um pouco de assembly e o conceito de pilha. Enquanto no stdcall a responsabilidade de desempilhar os argumentos ou valores é sempre da função chamada o fastcall tem os primeiros parâmetros armazenados nos registradores do processador enquanto que os subseqüentes ficam na pilha. Como mencionado a implementação de fastcall pode ser diferente de acordo com o compilador e não é aberta, pode ser proprietária. Na convenção stdcall os parâmetros são empilhados na ordem reversa, a função chamada desempilha todos os argumentos quando retorna, dados primitivos são retornados no registrador EAX ou nos registradores EAX:EDX e estruturas de até 8 bytes são retornadas no par EAX:EDX. Já o fastcall usa os registradores para os quatro primeiros argumentos e a pilha para os subsequentes. Entende-se por função chamadora a que está por fora, que faz a chamada e por função chamada aquela que é executada dentro de uma chamadora. Em um exemplo hipotético caso o Delphi usasse por padrão stdcall, se no evento onClick do button1 é executada a função ShowMessage('hello world') então a ShowMessage é a chamada e button1Click é a chamadora. ShowMessage teria a obrigação de desempilhar o parâmetro 'Hello World'.

Nota do DevMan

BorlandMM.dll é uma DLL que contém o gerenciamento de memória do Delphi. Nela são tratadas questões como alocação e liberação de memória usada por strings. Sharemem é uma unit que indica ao programa para fazer uso da BorlandMM.dll. Colocando-se a unit sharemem em uma DLL e no programa que fará uso dela é possível criar dentro da DLL funções que aceitam e devolvem strings nativas do Delphi, e também passar strings nativas por referência. Ao se criar uma nova DLL o Delphi coloca um comentário no arquivo .dpr explicando qual é o objetivo do BorlandMM, como e onde ele é usado. Para fins de interoperabilidade, chamadas a DLLs não compiladas em Delphi e chamadas a objetos COM ou .NET não se usa a BorlandMM porque as strings nativas do Delphi não são entendidas pelas outras linguagens. Em casos de interoperabilidade usa-se sempre PChar, PAnsiChar e PWideChar.

Listagem 1. DLL feita no Delphi

```
library DelphiDll;

uses
 sharemem,
 SysUtils,
 Classes,
 dialogs;

{$R *.res}

function Soma(x,y: Integer):Integer; stdcall;
begin
 Result := x+y;
end;

procedure EchoAnsiString(s: PAnsiChar); stdcall;
begin
 ShowMessage( string(AnsiString(s)) + ' --- ANSI de dentro da DLL ');
end;

procedure EchoWideString(s: PChar); stdcall;
begin
 ShowMessage( string(s) + ' --- WIDE de dentro da DLL ');
end;

exports
 Soma,
 EchoAnsiString,
 EchoWideString;

begin

end.
```

Listagem 2. Importando a DelphiDLL no Prism

```
MainForm = partial class(System.Windows.Forms.Form)
private
 method btSomar_Click(sender: System.Object; e: System.EventArgs);
 method btEcho_Click(sender: System.Object; e: System.EventArgs);
protected
 method Dispose(disposing: Boolean); override;
public
 constructor;
 [DllImport('DelphiDll.dll')]
 class method Soma(x, y: Integer): Integer; external;
 [DllImport('DelphiDll.dll')]
 class method EchoAnsiString(s: String); external;
 [DllImport('DelphiDll.dll', CharSet := CharSet.Unicode)]
 class method EchoWideString(s: WideString); external;
end;
```

Nota do DevMan

Strong Name – É como é chamado o identificador único do assembly. Essa identidade única é obtida através de seus atributos (linguagem, versão etc.). Esse strong name é representado por um arquivo .snk e para gerá-lo utiliza-se a ferramenta Sn.exe, que dependendo do .NET Framework ou das SDKs instaladas pode estar em um diretório diferente ou até mesmo haver várias versões dessa ferramenta instalada. Neste artigo foi usada a localizada em C:\Arquivos de programas\Microsoft SDKs\Windows\v6.0A\bin\sn.exe. Depois de gerar o arquivo .sn coloque-o no mesmo diretório que a solution e vá em Project>properties>signing e marque a opção "sign the assembly".Também é possível assinar o assembly com um arquivo pfx protegido por uma senha particular. Esses arquivos fornecem um par de chaves privada e pública para o assembly para que ele possa ser colocado no GAC. Depois de assinado e compilado basta abrir a pasta c:\windows\assembly e arrastar a DLL lá dentro.

Nota do DevMan

GAC, ou **Global Assembly Cache** é uma área do sistema operacional portador do .NET Framework onde podem ser colocados e registrados assemblies públicos para uso geral de vários aplicativos. O GAC permite que versões diferentes da mesma biblioteca coexistam em um mesmo espaço mesmo que possuam o mesmo nome, contanto que suas chaves e suas versões (Assembly Version) sejam diferentes. O GAC pode ser acessado em c:\windows\assembly, e embora ele pareça e se comporte como uma pasta, na verdade ele é um programa.

Exemplo prático - Usando DLLs do .NET no Delphi

Para chamar DLLs do .NET no Delphi é necessário que a propriedade ComVisible da biblioteca seja true. Além disso, é necessário também que a biblioteca do .NET tenha um strong name e que seja registrada no GAC (Global Assembly Cache).

A classe a ser chamada de dentro dessa biblioteca deve obrigatoriamente ser pública e deve obrigatoriamente implementar uma interface com GUID.

Resumidamente a biblioteca .NET inteira deve se comportar como um objeto COM e ser exportada como tal, para que o Delphi a importe da mesma maneira. Entretanto nesse exemplo não serão usados os assistentes de importação e exportação de Type Libraries, mas todas as referências serão feitas manualmente.

No .NET para que uma biblioteca seja Com Visible, ou seja, para que possa ser vista e importada pelo Delphi, VB6 ou qualquer outra linguagem que consome bibliotecas COM, é necessário que se edite o arquivo AssemblyInfo.cs (ou .pas) e se coloque a propriedade [assembly: ComVisible(true)]. Um exemplo de DLL feita no Delphi Prism pode ser visto na **Listagem 3.**

Algumas linhas devem ser acrescentadas no arquivo AssemblyInfo.pas:
[assembly: ComVisible(true)] - Para a biblioteca ser visível via COM;
[assembly: Guid("08D81D5B-0F82-4c88-AB8A-44EFA793B6C6")] - o GUID da biblioteca;
[assembly: AssemblyKeyFile ('TOperacoes.snk')] - Um arquivo keyfile criado na ferramenta sn.exe.

O arquivo AssemblyInfo merece atenção especial: a versão no C# é 1.0.0.0 e no Prism é 1.0.0.1. Essa versão é importante para à chamada a função que será vista na **Listagem 4**.

É muito importante neste exemplo que a classe TOperacoes implemente a interface MarshalByRefObject, além de implementar a interface IOperacoesInterface. MarshalByRefObject fará com que sua classe possa ser instanciada via COM. COM ou Component Object Model é um conjunto de interfaces e padrões criados pela Microsoft em 1993 para favorecer a intercomunicação e a interoperabilidade entre processos. Com o COM era possível criar classes instanciar objetos através de interfaces. Essas classes eram possíveis de criar em qualquer linguagem, ou seja, era um sistema de programação orientada a objetos quase independente de linguagem, onde qualquer linguagem poderia criar e compilar bibliotecas para uso COM seguindo certos padrões. As interfaces usadas no COM tinham GUIDs, identificadores únicos. O Delphi começou a ter suporte a GUIDs a partir da versão 2 para ter compatibilidade com o COM. Outras características do COM são os gerenciamentos de threads e de memória. Uma

das maneiras mais fáceis de assegurar a interoperabilidade entre o Delphi e bibliotecas .NET é tornar estas compatíveis, ou visíveis, ao COM. A integração Delphi – COM – .NET é mais fácil do que Delphi – .NET direto. O objeto .NET ser compatível com o COM também favorece sua integração com VB6 e outras linguagens.

Para uma aplicação Delphi instanciar um objeto do .NET ele deve estar registrado no GAC. A aplicação deve enxergar a DLL mscoree.dll, localizada na pasta system32, que concentra várias funções internas do .NET Framework. Esta biblioteca, mscoree.dll, é o coração nativo do .NET, onde o código não é gerenciado.

Entre seus vários métodos destaca-se o ClrCreateManagedInstance, que é capaz de criar e devolver uma referência a uma instância de uma classe qualquer do .NET.

Os parâmetros para ClrCreateManagedInstance são: "nome completo" da classe, Interface que combine com a classe (não é uma variável dessa interface, mas a interface em si) e o último parâmetro é sim uma variável do tipo da interface do parâmetro anterior. Este último parâmetro é que receberá uma instância da classe. Como ele é tratado como interface no Delphi não é necessário se preocupar com a destruição dele. Além da contagem de referências temos o Garbage Collector do .NET.

Garbage Collector é o mecanismo "coletor de lixo" do .NET Framework. Similar ao Java e a outras linguagens gerenciadas, o Garbage Collector permite que o programador não se preocupe com detalhes de gerenciamento de memória. O programador pode instanciar objetos sem se preocupar com a destruição deles ou com vazamentos de memória. O coletor de lixo se encarregará de verificar se os objetos estão ou não sendo usados, se há ou não referências a eles e caso não haja, ele destrói os objetos e libera a memória.

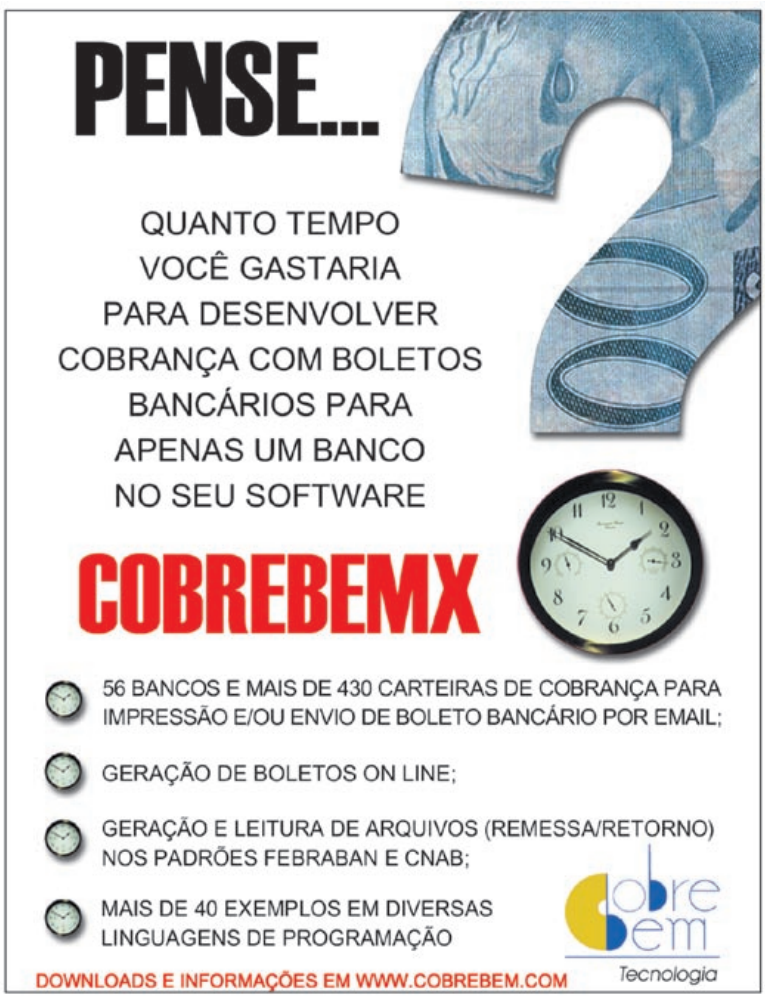
Por nome completo entenda-se um qualificador completo do assembly, incluindo Namespace.Classe, Classe, Versão, Cultura, chave pública. Na **Listagem 4** é mostrado o exemplo com TOperacoes. É absolutamente necessário criar do lado Delphi uma interface idêntica à criada dentro da biblioteca .NET e com o mesmo GUID. Dentro do Delphi Prism as strings são WideStrings. Por isso a interface usará WideStrings também.

Na **Listagem 4** também fica evidente a necessidade das units ActiveX e ComObj no uses. É por causa do uso das funções OleCheck e do tipo TIID. OleCheck verifica se o HResult HR retornado por ClrCreateManagedInstance é um HResult válido. Caso não seja válido OleCheck dispara uma exceção que ajuda a alertar que o parâmetro ppObject, que deveria vir com uma instância da classe .NET, por algum motivo estará apontando para um endereço inválido ou nil.

O botão do exemplo cria a instância TOperacoes de dentro do .NET retornando essa instância na variável op do tipo IOperacoesInterface e executa, mostrando com ShowMessage, os três métodos de op. Existem componentes e bibliotecas

Listagem 3. Criando uma DLL no Delphi Prism

```
namespace Operacoes;

interface

uses
  System,
  System.Runtime.InteropServices;

type
  [Guid("7D0B0046-79C0-4c48-B9E4-C0ECB9732891"),
    InterfaceType(ComInterfaceType.InterfaceIsIUnknown)]
  IOperacoesInterface = public interface
    method Soma(x, y: Integer): Integer;
    method EchoString(s: String): String;
    method UmaString: String;
  end;

  TOperacoes = public class(MarshalByRefObject, IOperacoesInterface)
  private
  protected
  public
    method Soma(x, y: Integer): Integer;
    method EchoString(s: String): String;
    method UmaString: String;
  end;

end.
```

Listagem 4. Consumindo pelo Delphi a DLL feita em Prism

```
unit Unit1;
interface
uses
  Windows, Messages, SysUtils, Variants,
  Classes, Graphics, Controls, Forms,
  Dialogs, ActiveX, StdCtrls, ComObj;

type
  TForm1 = class(TForm)
    Button1: TButton;
    procedure Button1Click(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;
  IOperacoesInterface = interface(IUnknown)
  ['{7D0B0046-79C0-4c48-B9E4-C0ECB9732891}']
    function Soma (x, y: integer): integer; safecall;
    function EchoString(s: WideString): WideString; safecall;  //esta é a parte mais
    interessante: a mensagem deve ir como string, mas voltar como pchar
    function UmaString: WideString; safecall;
  end;

var
  Form1: TForm1;
function ClrCreateManagedInstance(pTypeName: PWideChar; const riid: TIID;
    out ppObject): HRESULT; stdcall; external 'mscoree.dll';
implementation
{$R *.dfm}
procedure TForm1.Button1Click(Sender: TObject);
var
    hr: HRESULT;
    op: IOperacoesInterface;
    ClasseDoDotNet: string;
begin
    ClasseDoDotNet := 'Operacoes.clsOperacoes, clsOperacoes,
    Version=1.0.0.0, Culture=neutral, PublicKeyToken=f252bd371a42685b';
    hr := ClrCreateManagedInstance(PWideChar(ClasseDoDotNet),
    IOperacoesInterface, op);
    OleCheck(hr);
  ShowMessage(inttostr(op.Soma(8,5)));
  ShowMessage(op.UmaString);
  ShowMessage(op.EchoString('Hello World'));
end;
end.
```

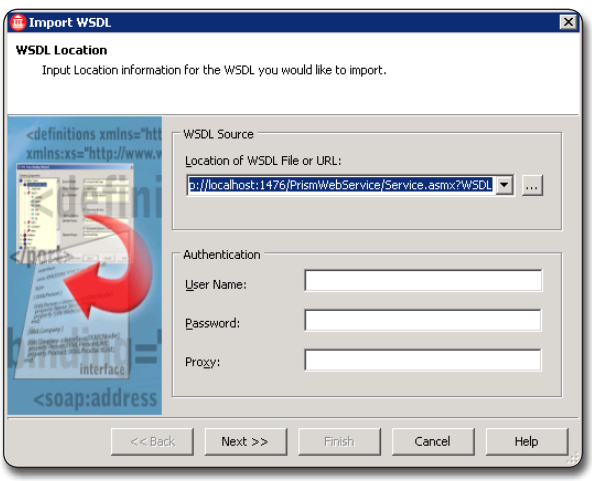


Figura 1. O WSDL Importer

para facilitar o consumo de bibliotecas do .NET a partir do Delphi. Um exemplo é o Hydra, da RemObjects, mesma empresa responsável pelo Delphi Prism e pelo Pascal Script. O outro componente é o CrossTalk da Atozed, mesma empresa do Indy e do Final Builder.

Muita atenção com relação às strings. Embora já foi extensamente discutido que as novas strings do Delphi XE são strings onde cada char tem 2 bytes (WideChar) a string Unicode nativa do Delphi XE não é WideString, ela é BASEADA em WideString, ou COMPATÍVEL com WideString. A diferença entre uma string do Delphi e uma WideString tradicional é que enquanto a WideString tradicional é uma string do tipo BSTR ou ponteiro para WideChar do Windows, a nova string nativa, UnicodeString, tem antes do primeiro caractere espaços reservados para CodePage, ElementSize, RefCount e Length. WideStrings não têm contagem de referências, por exemplo.

Veja o link para o artigo "Unicode in RAD Studio" na seção de links. Abaixo do subtítulo WideString a Embarcadero deixa bem claro que para uso com o COM, e consequentemente .NET, é recomendável o uso de WideString.

Se por acaso for usado string no lugar de WideString a string que viria como resultado de uma chamada a uma biblioteca .NET pode vir truncada por um motivo simples: O Delphi receberá uma string no formato "vetor de caracteres terminado em #0" e entenderá como string. Tentará usar os 12 primeiros bytes como CodePage, ElementSize, RefCount e Length, e por isso vai perder o verdadeiro tamanho e formato da string.

O caso contrário (passagem de string de uma DLL em Delphi para o .NET) não acontece porque não é sequer possível passar os tipos String, WideString, AnsiString ou UnicodeString, tendo que ser substituídos por PChar ou PAnsiChar dependendo do caso.

Qualquer tentativa de passar uma string de uma DLL em Delphi para um programa em .NET resultará em um Access Violation, dentre vários motivos, porque a memória usada para alocar essa string é criada dentro da DLL e o programa .NET, além de não ter acesso a essa área de memória, não conhece o formato da string no Delphi. A passagem de strings de uma DLL .NET para o Delphi não gera um Access Violation na maioria das vezes, mas pode truncar a string.

Exemplo prático - Consumindo Web Services do Prism no Delphi

Para isso é necessário criar os Web Services no Delphi Prism e iniciar o servidor, podendo ser o IIS ou o servidor interno do Visual Studio utilizado para debug. Depois abrindo o Delphi basta usar o WSDL Importer (**Figura 1**) para criar um client para esse Web Service.

A unit e a classe do Delphi terão o mesmo nome que o arquivo .asmx gerado no Prism. A **Listagem 5** mostra o código do Web Service e **Listagem 6** o código para consumi-lo. O Delphi permite que se crie o objeto SOAP muito facilmente, que é o

wrapper / Proxy para o Web Service. Para consumir o Web Service pelo Delphi, foi criado um formulário com apenas um botão e um controle TMemo. A ideia é o botão chamar os métodos e registrar no Memo os retornos. Uma vez feita a referência ao Web Service, basta instanciar o referido objeto.

O interessante deste exemplo é que é usado um tipo estruturado, que é o tipo "contato". Ele é um Record. O método CriaContato aceita os parâmetros necessários para se criar um contato, cria a variável dentro do Web Service (.NET) e a retorna. Porém um fato merece atenção: Embora o tipo contato seja um Record do lado .NET, no lado Delphi ele é representado por uma classe descendente da classe TRemotable. Cada membro do Record contato do lado Prism é mapeado para uma propriedade da classe contato do lado Delphi.

Se do lado Delphi contato não é um Record e sim uma classe, o retorno de CriaContato também não é um Record, mas sim um objeto instanciado da classe contato. Por isso, para evitar memory leaks, é necessário destruir contato após o uso.

O método EchoContato aceita um contato como parâmetro e o modifica no lado servidor, devolvendo o objeto com os rótulos textuais "Nome...:", "Email...:" e "Telefone...:" nas respectivas propriedades.

Obviamente esses rótulos são desnecessários, foram colocados aí com um propósito duplo: mostrar que é possível passar um objeto para o servidor, modificá-lo e retornar o objeto modificado e também demonstrar que o método EchoContato ilustra bem o fato de contato ser objeto, porque mesmo o modificador do parâmetro sendo const o contato é passado como se fosse por referência (assim como todo objeto ou ponteiro) e retorna modificado para o Delphi.

O método estático GetServiceSoap cria um objeto do tipo do seu serviço, dispo-

Nota do DevMan

Um Proxy, na programação, é um padrão de projeto estrutural que serve de interface para uma outra classe ou para bibliotecas de código não orientado a objetos. Todas as classes que precisam das funcionalidades ou serviços de uma outra classe real podem utilizar o Proxy. O Proxy fica com a responsabilidade de internamente fazer o uso correto da classe real. A classe real pode ser trocada, ou ter toda a sua implementação modificada, que o Proxy continua o mesmo e a aplicação continua compilando. Um exemplo de Proxy seria uma classe TCompressor com um método comprimir(nomearquivo: string). Esta classe seria bem enxuta, mas dentro do método comprimir haveria a responsabilidade de se instanciar objetos de compressão padrão (zlib, Jedi, Zip, Rar) para comprimir o arquivo. Mesmo que isso fosse mudado, a classe Proxy TCompressor é tão simples que qualquer objeto que a use continuaria podendo ordenar a compressão de um arquivo de forma transparente. Um outro exemplo de Proxy são os componentes e classes da paleta Indy. Eles são proxies para a DLL Winsock.dll da API do Windows. A responsabilidade de estabelecer conexões TCP/IP e transferir dados é desta DLL, mas os componentes da Indy tornam essa tarefa mais transparente para o programador de alto nível. Outro exemplo de Proxy seria uma classe TEnvio com um método enviar(nomearquivo: string). Dependendo do contexto, o método enviar poderia "envelopar" instâncias locais dos componentes da Indy (ou outros) para enviar o arquivo tanto via e-mail como FTP ou HTTP. Se toda a implementação do método Enviar fosse trocada usando-se outras classes ainda assim a classe TEnvio poderia ser usada transparentemente por outras classes. As classes geradas automaticamente pelo WSDL importer são proxies porque embora elas simulem uma classe normal, representando seus métodos e propriedades e implementando uma interface idêntica à que é implementada no servidor, a classe gerada pelo WSDL esconde do programador final uma série de passos muito mais complicados do que ele imagina. Quando um método é executado um pacote SOAP, um formato de XML próprio para Web Services é montado, uma conexão HTTP é estabelecida com o servidor no endereço especificado, o XML do SOAP é enviado via método POST do protocolo HTTP. O Servidor responde com outro envelope SOAP que é analisado pelo framework de Web Services do Delphi com a resposta do que foi executado no servidor, seja um retorno de função, sejam variáveis passadas por referência. Para o programador parece que ele está simplesmente chamando um método.

Listagem 5. Web Service feito no Prism

```
type
  contato = public record
  private
  public
    Nome: string;
    Email: string;
    Telefone: string;
  end;
  [WebService(&Namespace := 'http://tempuri.org/')]
  [WebServiceBinding(ConformsTo := WsiProfiles.BasicProfile1_1)]
  Service = public class(System.Web.Services.WebService)
  public
    method HelloWorld: string;
    method Soma(x, y: Integer): integer;
    method EchoString(s: string): string;
    method CriaContato(nome, email, telefone: string): contato;
    method EchoContato(var c: contato);
  end;
implementation
[WebMethod]
method Service.HelloWorld: string;
begin
  result := 'Hello World';
end;
//implementação dos outros métodos disponível no download
end.
```

Listagem 6. Consumindo Web Service feito no Prism

```
procedure TForm1.btChamarMetodosClick(Sender: TObject);
var
  serv: ServiceSoap;
  cont: contato;
begin
  serv := GetServiceSoap();
  mMetodos.Lines.Add(serv.HelloWorld);
  mMetodos.Lines.Add(serv.EchoString('Teste!'));
  mMetodos.Lines.Add( inttostr( serv.Soma(8,9)));

  cont := serv.CriaContato('Vitor', 'vitorrubio@gmail.com', '12345678');
  mMetodos.Lines.Add('....'+cont.Nome);
  mMetodos.Lines.Add('....'+cont.Email);
  mMetodos.Lines.Add('....'+cont.Telefone);

  serv.EchoContato(cont);

  mMetodos.Lines.Add('....'+cont.Nome);
  mMetodos.Lines.Add('....'+cont.Email);
  mMetodos.Lines.Add('....'+cont.Telefone);

  cont.Free;
end;
```

nibilizando seus métodos. Se o projeto consome vários Web Services diferentes então deve ser usado o formato NomeDaUnit.GetServiceSoap.

O parâmetros de GetServiceSoap são opcionais, mas é possível especificar o endereço do Web Service e se será utilizado algum componente THTTPRIO diferente/personalizado da paleta Web Services.

Nota

Olhando-se seção initialization da unit Service.pas (ou a unit que representa o Web Service importado) nota-se a linha InvRegistry.RegisterInvokeOptions (TypeInfo (ServiceSoap), ioDocument) onde ServiceSoap é o nome do serviço seguido da palavra Soap. No Delphi 7 essa linha não é adicionada automaticamente. Ela deve ser adicionada caso o Web Service a ser consumido seja feito em .NET Framework 2.0 ou superior, mas deve ser omitida caso o Web Service seja feito em .NET 1.1. O motivo disso é que a maneira (chamada binding) como os parâmetros de entrada e saída do envelope SOAP do Web Service .NET são especificados mudou da versão 1.1 para a 2.0. Antes os parâmetros eram especificados em nós literais, agora são especificados dentro de nós Input/Output.

Listagem 7. Receptor feito em Delphi Win32

```
type
 TForm1 = class(TForm)
  Memo1: TMemo;
 private
  { Private declarations }

  procedure WMCopyData(var Msg: TWMCopyData);
  message WM_COPYDATA;
 public
  { Public declarations }
 end;

var
 Form1: TForm1;

implementation

{$R *.dfm}

{ TForm1 }

procedure TForm1.WMCopyData(var Msg: TWMCopyData);
var
 msgRecebida: string;
 i: integer;
begin
 msgRecebida := PChar(Msg.CopyDataStruct.lpData);
 Msg.Result := 1;
 memo1.Lines.Add(msgRecebida);
end;

end.
```

Exemplo prático - Troca de mensagens usando a API do Windows

Outra maneira, dentre muitas, de se fazer com que dois programas feitos para ambientes diferentes se comuniquem é pela API do Windows. Usando a função SendMessage e a mensagem WM_COPYDATA pode-se enviar para uma outra aplicação um ponteiro para uma estrutura de dados que "carregue" uma string. Basta que a outra aplicação monitore a message WM_COPYDATA para que automaticamente seja disparado um método que receberá o ponteiro para a estrutura e poderá trabalhar com seus campos.

Um dos desafios é que o ambiente gerenciado do .NET não possui e não usa explicitamente toda a API do Windows como faz o Delphi Win32. Internamente, nos fontes do framework, até são usadas as APIs do Windows, mas de forma transparente para o programador de alto nível. Essas são as APIs para se trabalhar com janelas, eventos e I/O, por exemplo. Nem todas as APIs são usadas. Algumas estão aos poucos sendo substituídas por bibliotecas gerenciadas.

O ambiente gerenciado do .NET não permite algumas coisas como ponteiros para PChar serem usados como integer (o que deveria porque os ponteiros tem 4 bytes) ou o uso de ponteiros genéricos. Por isso deve-se usar no .NET tipos que conhecidamente são, se portam ou possam ser serializados em forma de ponteiros para as DLLs da API do Windows. Quem faz o serviço de carregar e chamar as DLLs é o namespace System.Runtime.InteropServices.

Nos casos em que a API em questão solicita um ponteiro simplesmente porque o conteúdo da variável que está sendo passada como parâmetro vai ser modificado após o término da função pode-se passar por referência em vez de passar um ponteiro. Na verdade ponteiros são necessários no C e C++ quando se quer passar um valor por referência. No Delphi tem-se a palavra chave var que pode substituir a necessidade de ponteiros.

No caso do .NET não é permitido se passar ponteiros para estruturas empacotadas pelo gerenciador, como structs por exemplo. Por isso nesses casos usa-se a palavra var.

Na **Listagem 7** é mostrado o programa receptor. Basta preparar uma janela com um memo para que as mensagens sejam visualizadas e preparar uma procedure pública que monitore a mensagem COPYDATA do Windows. Dentro dela é possível pegar o parâmetro msg e fazer o que quiser com ele, não esquecendo de fazer msg.Result := 1 para avisar a aplicação que enviou a mensagem que tudo ocorreu bem.

O parâmetro msg é uma string, mas é aí que entra a flexibilidade do negócio. Por ser string ela pode transportar qualquer tipo de dado. Objetos e estruturas complexas podem ser serializados em forma de XML, passados para outra aplicação dessa forma e serializados em outro objeto similar, ou lidos diretamente com classes para manipulação de XML.

XMLs longos, com grandes vetores ou coleções de dados podem ser partidos, ou os nós podem ser separados cada um em um XML diferente e passados dentro de um loop em várias mensagens.

A procedure WMCopyData monitora a mensagem do Windows WM_CopyData. Sempre que essa mensagem for disparada e detectada, a procedure entra em ação executando seu código. Em sua implementação essa procedure obtém a string pelo membro lpData da estrutura TWMCopyData.CopyDataStruct e o converte para o tipo PChar. Também retorna do objeto Msg o retorno 1, indicando que tudo ocorreu bem.

A **Listagem 8** apresenta o emissor feito em Delphi Prism. Neste exemplo são usados AnsiStrings, mas tudo dependerá de qual API se usa e de que tipo de dados se manda. Como o Delphi Prism não possui uma "unit Windows" com o porte da API do Windows, foi necessário recriar algumas coisas como a estrutura TCOPYDATASTRUCT e a constante WM_COPYDATA que nada mais é que um número

inteiro positivo e constante. Todas as flags da API do Windows são numéricas.

O código mostra o uso de Handle. Handle (do inglês "manipular", "manusear", numa tradução livre) é um número inteiro não negativo diferente de zero que serve para identificar recursos na API do Windows. Esses recursos podem ser arquivos, identificadores de threads e identificadores de janelas. Como identificador de janelas ele serve para manipular as mesmas. Toda janela tem um handle. Quando uma janela abre dentro de outra, como um MDI Child, é porque foi dado um comando para que a janela de dentro tivesse como parent handle (handle pai) o handle da janela de fora. Handles são utilizados também na criação de janelas modais ou para definir que janela abriu qual janela, numa espécie de cadeia. Um button dentro de um panel tem um handle próprio e tem como parent handle o handle do panel, que por sua vez é filho do formulário onde se encontra. Do ponto de vista da API do Windows todos os Widgets, ou componentes gráficos, como botões e painéis têm um handle e são tratados como janelas. Um botão é, nos bastidores, uma janela para o Windows.

A função FindWindow tentará encontrar o handle de uma janela baseada no nome de sua barra de caption e de sua classe. O conceito de classe nas janelas da API do Windows não é o mesmo conceito de classe da POO, mas para facilitar e padronizar o Delphi cria janelas cujo nome de classe é o mesmo nome da classe do formulário que está no .pas.

Se a função FindWindow não encontrar a janela ela retorna zero. O artigo sobre API do Windows da ClubeDelphi 116 mostra em detalhes como usar FindWindow para encontrar por exemplo a calculadora do Windows, se estiver sendo executada.

A função SendMessage faz exatamente isso: envia mensagens. Essas mensagens são parâmetros, estruturas e flags da API do Windows. Ela tem uma gama muito grande de funções e objetivos que variam de acordo com quem são o remetente e o destinatário e quais são os parâmetros. Ela pode ser usada tanto para simular o pressionamento de uma tecla sem o uso do teclado como para um programa avisar outro para que coloque sua janela na frente das outras. Nos exemplos desse artigo SendMessage está sendo usado para enviar de um programa para outro, através dos handles de suas janelas, um ponteiro para uma estrutura de dados que contém uma string. Um pchar na verdade. A mensagem em questão é enviada nesse pchar, e poderia ser um objeto inteiro serializado em XML.

Os argumentos de SendMessage são na realidade todos ponteiros, e por isso mapeados no Delphi como DWORD ou Cardinal, que são inteiros sem sinal de 4 bytes. Ponteiros devem ter seu valor numérico expresso em inteiros sem sinal porque não existe endereço de memória negativo.

No .NET não é permitido usar DWORD ou UInt32 como ponteiros genéricos, nem usar ponteiros não tipados ou ponteiros de um tipo diferente que a função aceite. Por isso devem ser criados overrides (assinaturas) diferentes para cada possível uso de SendMessage. No exemplo o SendMessage recebe os parâmetros intPtr, Integer, intPtr e um TCOPYDATASTRUCT passado por referência.

A documentação completa da função SendMessage encontra-se no site MSDN.

Listagem 8. Receptor feito em Delphi Prism

```
type
 TCOPYDATASTRUCT = record
  dwData: IntPtr;
  cbData: Integer;
  lpData: string;
 end;

 MainForm = partial class(System.Windows.Forms.Form)
 private
   method button1_Click(sender: System.Object; e: System.EventArgs); unsafe;
 protected
  method Dispose(disposing: Boolean); override;
 public
  constructor;

  const WM_COPYDATA = $004A;

  [DllImport("user32.dll")]
  class method SendMessage(
   hWnd: IntPtr; Msg: Integer; wParam: IntPtr;
   var lParam: TCOPYDATASTRUCT): IntPtr; external; unsafe;

  [DllImport("user32.dll")]
  class method FindWindow(
   lpClassName: string; lpWindowName: string): IntPtr; external;
 end;

implementation

method MainForm.button1_Click(sender: System.Object; e: System.EventArgs);
var
 //handle da janela receptora
 hw: integer;
 copydata: TCOPYDATASTRUCT;
begin
 hw := FindWindow('TfrmReceptor','Receptor');

 copydata.dwData := 0;
 copydata.cbData := 1 + txtMensagem.Text.Length;
 copydata.lpData := txtMensagem.Text + #0;

 if ( SendMessage(hw, WM_COPYDATA, Longint(self.Handle), var copydata ) <> 1) then
 begin
  MessageBox.Show('Impossível enviar a mensagem');
 end;
end;
```

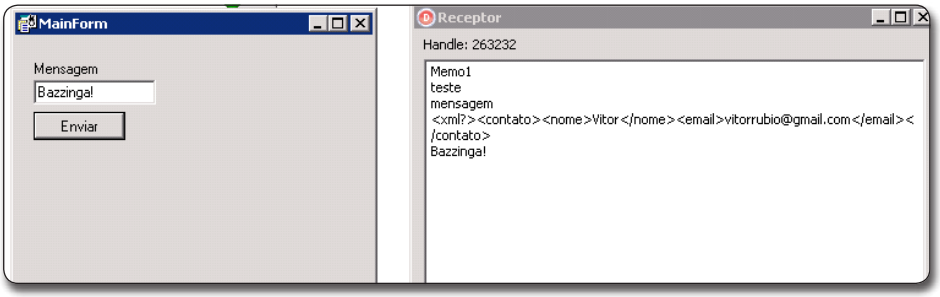


Figura 2. Emissor e receptor conversando

Na unit Windows.pas do Delphi é possível ver uma lista de algumas mensagens que ela suporta. Todas são constantes numéricas começadas com as letras WM.

Executando os dois programas pode-se digitar uma mensagem no TextBox do programa feito em Delphi Prism e essa mensagem aparecerá no Memo do programa feito em Delphi Win32 (**Figura 2**).

A mensagem a ser processada pode ser um cálculo longo, uma solicitação de relatório ou uma gravação em banco de dados. O programa receptor pode fazer uma query trazendo uma quantidade grande de dados e mandar o relatório para o solicitante por e-mail, sem comprometer a performance do programa emissor. O programa emissor pode enviar essa mensagem em uma thread à parte, se necessário. As possibilidades são muitas.

Conclusão

Este artigo não é um guia, tabela de referência ou pesquisa metódica sobre interoperabilidade. Para que fosse seria necessário usar o mesmo exemplo para todas as integrações e os exemplos deveriam ter transferências de todos os tipos de dados simples / elementares, mostrando um "de – para" nas duas pontas da integração. Além disso, deveria também ter transferências com objetos, interfaces e objetos aninhados.

Muitas dificuldades são encontradas no processo de integração, e este artigo serve como ponto de partida para que os mesmos erros básicos não sejam repetidos em todos os projetos de integração, e para que as dúvidas básicas sejam sanadas.

A integração e interoperabilidade é uma área que sempre terá necessidades de profissionais "multidisciplinares" e sempre haverá trabalho.

Nas integrações que não são online (batch ou lote, com um horário agendado) tenha a preferência por arquivos XML puros (não no padrão data packet) em detrimento de arquivos posicionais ou com separadores. Isso fará com que o arquivo seja válido mesmo que seja adicionado um novo campo null. Além disso, todas as linguagens hoje têm recursos para tratar o arquivo XML como um datasource ou um vetor de objetos, dependendo do caso.

Tenha preferência por Web Services. Páginas que aceitam uma requisição HTTP pura e devolvem um XML puro, sem envio e recebimento de envelopes SOAP podem ser, teoricamente, considerados Web Services, assim como aplicações REST.

O uso de DLLs é recomendado para funções de baixo nível, ou pelo menos de um nível mais baixo que a camada de negócios e de persistência. Por exemplo, uma biblioteca de funções. Na maioria dos casos as DLLs são boas para serem usadas como bibliotecas auxiliares externas que prestam algum serviço esporádico como um cálculo complexo ou uma criptografia. Não é recomendável fazer uma camada inteira do sistema sobre DLLs. Elas devem ser feitas como se estivessem sendo feitas em C/C++, com chamadas stdcall, para que sejam usadas em qualquer linguagem.

Bibliotecas .NET podem ser usadas pelo Delphi Win32 (e até por outras linguagens, gerenciadas ou não), mas envolve um custo e trabalho altos além de curva de aprendizado acentuada. Trabalhando-se com bibliotecas .NET deve-se dar preferência a integrar com outros programas / bibliotecas também .NET ou com Web Services.

Pode ser interessante encapsular chamadas a Web Services dentro de DLLs para que essas possam ser consumidas por aplicações que não se utilizam de Web Services nativamente, como linguagens de script client side, Excel, Access, e sistemas ABAP, Progress, Natural, Cobol ADVPL etc. Embora muitas dessas linguagens/ferramentas possuam módulos para consumir Web Services, a maioria das vezes não são módulos nativos, devem ser comprados à parte e muitas vezes o custo se torna proibitivo.

Este artigo não abordou outras formas de interoperabilidade muito utilizadas como COM, COM+, DCOM, ActiveX entre outras. As possibilidades são imensas, por isso deixamos a cargo do leitor pesquisar novas possibilidades conforme a necessidade.

Vitor Luiz Rubio
vitorrubio@gmail.com
http://www.vitorrubio.blogspot.com
Tecnólogo em Processamento de Dados pela FATEC-SP e MCP. Atua como Analista de Sistemas Sênior na Editora Revista dos Tribunais. Programa com Delphi desde a versão 3 e com .Net desde a versão 1.1.

Blogs do autor
http://www.vitorrubio.blogspot.com/
http://twitter.com/vitorrubio

Unicode in RAD Studio
http://docwiki.embarcadero.com/RADStudio/XE/en/Unicode_in_RAD_Studio

Hydra
http://www.remobjects.com/hydra/

Atozed Crosstalk
http://www.atozed.com/crosstalk/index.EN.aspx

Managed-VCL
http://www.managed-vcl.com/

Dê seu feedback sobre esta edição!

A Clubedelphi tem que ser feita ao seu gosto. Para isso, precisamos saber o que você, leitor, acha da revista!

Dê seu voto sobre este artigo, através do link:

www.devmedia.com.br/clubedelphi/feedback

Trace API do Firebird 2.5

Faça a auditoria do que acontece com o banco de dados

Jederson Zuchi

Resumo DevMan

De que trata o artigo:

Este artigo tem por objetivo demonstrar os recursos da Trace API, que foram introduzidos no Firebird 2.5. Será apresentado um exemplo prático em Delphi que monitora o BD.

Para que serve:

Serve para monitorar o que está acontecendo em tempo real no banco de dados. Desta forma consegue-se ver comandos e situações que ocorrem na base de dados, e assim, conseguir detectar problemas e tentar corrigi-los.

Em que situação o tema é útil:

Este tema é útil para fazermos auditorias na base de dados. É mais uma ferramenta disponível para que possamos aperfeiçoar cada vez mais nossos softwares. Neste artigo iremos apresentar alguns comandos SQL e algumas transações e ver como essa API pode nos ajudar a entender o que acontece no Banco de Dados.

Trace API:

O Firebird tem evoluído cada vez mais. Em sua versão 2.1 foram introduzidas as tabelas de monitoramento e agora em sua versão 2.5 uma nova API foi introduzida: a Trace API. Essas são ferramentas de diagnóstico que permitem ao desenvolvedor/DBA prever problemas em bases de dados, e assim garantir o bom funcionamento de sistemas que as utilizem. Este artigo se destina a explorar as novidades trazidas pela Trace API. Um aplicativo será criado para ilustrar sua utilização e obtenção de resultados.

Em um banco de dados ocorrem diversas situações por baixo de simples comandos DML e também DDL que não somos capazes de observar. Por exemplo, quando se digita um simples select * from <tabela>, o banco de dados tem que se encarregar de recuperar quais são as colunas da tabela para aí sim poder trazer o resultado. Isso acontece porque no comando Select foi especificado o caractere *. Parece uma bobeira, mas quando se usa sistemas que fazem uso de meios de comunicação como a internet, os dados estão sujeitos à velocidade da banda e quanto mais dados consegue-se economizar maior ficará a velocidade com que se irá obter uma determinada informação. No entanto, para conseguir otimizar o tempo de acesso para estas informações, tem-se que ter ferramentas que mostram os dados conforme eles vão acontecendo e é neste ponto que entra uma novidade do banco de dados Firebird 2.5, a Trace API.

A Trace API surgiu, porém foi pouco explorada. Com ela é possível monitorar e auditar um banco de dados conforme eventos ocorrem. Ainda nessa versão ela é um plug-in do banco de dados e roda do lado do servidor. As bibliotecas responsáveis por fazer essa auditória são a fbtrace.dll no Windows e fbtrace.so no Linux e sistemas Posix. Existem algumas ferramentas no mercado também que fazem uso dessa nova funcionalidade, inclusive ferramentas brasileiras.

Um ponto importante a destacar, é que a Trace API diferencia-se das tabelas de monitoramento que o Firebird traz juntamente com o banco de dados desde a versão 2.1. A principal diferença entre elas, é que a Trace API funciona como se fosse um log sequencial dos dados, desta forma ela registra a sequência dos eventos em sua ordem cronológica enquanto que as tabelas de monitoramento funcionam como uma fotocópia dos dados que estão ocorrendo no banco, como por exemplo: quais transações estão abertas, quais são os usuários, qual o caminho do banco de dados, page size definido na base de dados etc.

Para exemplificar imagina-se uma situação aonde um determinado sistema abre uma transação. Se a Trace API estiver sendo executada com a opção de verificar o início das transações, ela irá mostrar que uma transação é começada, em seguida as tabelas de monitoramento terão acesso a ela. E quando a transação é terminada, ela sai da tabela de monitoramento e é gerado outro evento na Trace API indicando que uma determinada transação foi finalizada.

Para fazer uso da Trace API tem-se que configurar um arquivo que vem junto com a instalação do banco de dados que se chama FBTrace.conf. Dentro do FBTrace.conf configura-se alguns parâmetros tais como mostra a **Tabela 1**.

É importante o conhecimento deste arquivo para que se consiga auditar somente aquilo que realmente é de interesse no momento. No aplicativo de demonstração será possível verificar alguns exemplos de geração desse arquivo e verificar os

resultados obtidos, depois que o trace rodar com os parâmetros selecionados no momento.

Depois que o arquivo de configuração foi configurado, há um executável que faz a captura dos logs que são gerados no banco de dados, o comando em questão é o fbtracemgr que já vem junto com a instalação padrão do Firebird 2.5. O FBTraceMgr requer alguns parâmetros necessários para seu funcionamento, e de acordo com cada parâmetro selecionado outros se tornam obrigatórios. A **Figura 1** exibe esses parâmetros.

No entanto, o comando mais utilizado para fazer o início da trace é o comando:

```
fbtracemgr –SE <IP_do_servidor_firebird>:service_mgr
–START –NAME <nome_do_trace> -CONFIG <local_do_
arquivo_de_configuracao> -USER <usuário_banco>
-password <Senha_banco>
```

Para finalizar o comando pode-se digitar conjuntamente CTRL+C ou abrir em outro prompt (ou shell no caso do Linux) o comando:

```
fbtracemgr –SE <IP_do_servidor_firebird>:service_mgr
–STOP –ID <id_do_trace> -USER SYSDBA –password
masterkey
```

Um detalhe a salientar é que o ID é fornecido no momento que é iniciada a Trace API. Este Id é um número sequencial que tem início em 1 e vai sendo incrementado à medida que se realizam as auditorias no banco. Este ID é somente zerado quando o servidor firebird é parado e iniciado novamente.

Exemplo do comando

No exemplo a seguir, configurou-se o arquivo de configuração para fazer uma auditoria das conexões que um determinado servidor Firebird 2.5 obtiver. No arquivo foram configuradas somente as opções enabled da seção database para true e a opção log_connections como true, todas as outras opções como false. Para fazer a chamada do comando FbtraceMgr usou-se a seguinte sintaxe:

```
fbtracemgr –SE service_mgr –START –NAME exemplo_1
–CONFIG fbtrace.conf –USER SYSDBA –PASS masterkey
```

A opção –SE define o serviço que será utilizado, que neste caso é a própria API do Firebird seguida da opção de –START que inicia a auditoria. –NAME vai definir qual é o nome desta sessão de auditoria, a opção –CONFIG recebe o arquivo de configuração, -USER define qual é o usuário que será dono dessa sessão de auditoria e o –PASS (ou -password) a senha. Com esta chamada tem-se o resultado da **Figura 2**. Vale ressaltar novamente, que

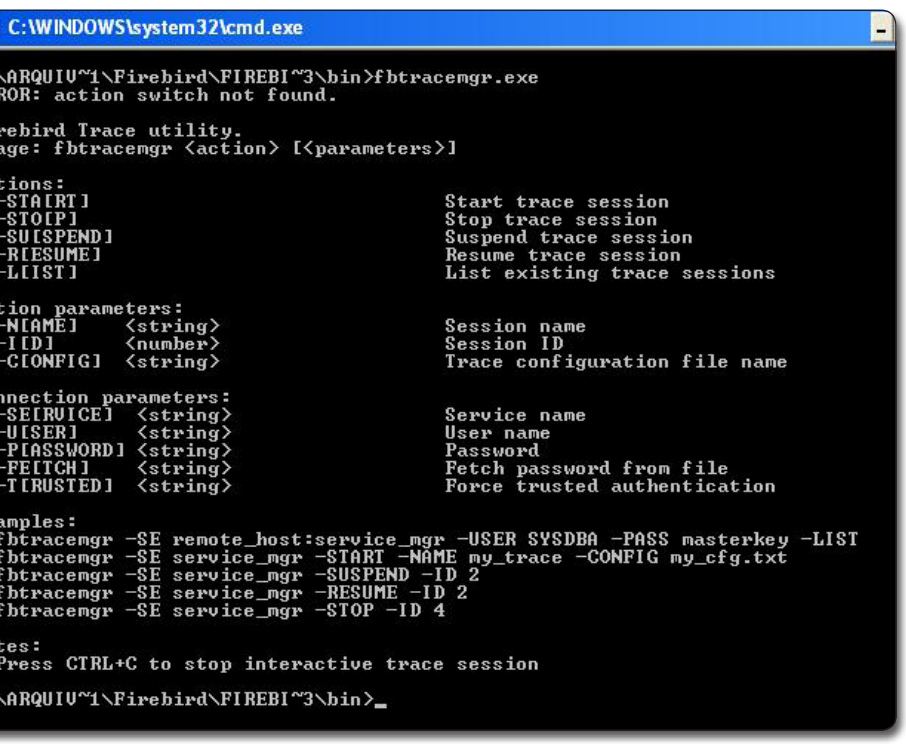


Figura 1. Lista de parâmetros do comando FbTraceMgr

Nota do DevMan

As tabelas de monitoramento do Firebird foram uma das novidades que surgiram na versão 2.1 do Firebird. Vale destacar que estas tabelas fazem o monitoramento do que está acontecendo na base de dados. É bastante útil em casos onde é preciso avaliar o que uma determinada base de dados está executando, e para isso basta que seja executado um simples comando SELECT em uma das tabelas de monitoramento para se saber o que está se passando naquele momento no banco de dados. As tabelas existentes são:

MON$DATABASE: É utilizada para verificar quais são as base de dados que estão sendo usadas naquele momento, e também para mostrar algumas informações tais como caminho da base de dados, tamanho do page_size, próxima transação, transação atual em aberto, tipo do dialeto que a base foi criada, dentre algumas outras informações relativas a base de dados.

MON$ATTACHMENTS: Esta tabela diz respeito às conexões que estão conectadas no Sistema Gerenciador de Banco de dados. Algumas informações pertinentes a esta tabela dizem respeito ao ID do processo no sistema operacional, o status da conexão se está em espera ou executando, o usuário que está conectado nesta seção, qual o protocolo que está sendo usado para manter esta conexão, o endereço da máquina que está com a conexão em aberto, etc.

MON$TRANSACTIONS: É utilizada para verificar o estado das transações abertas em uma determinada base de dados, algumas informações que são possíveis resgatar nesta tabela são o tipo de isolamento da transação, se ela está ativa ou em espera, se ela está em lock ou não. Um ponto importante a destacar é que esta tabela está vinculada a uma determinada conexão, podendo saber em específico quais são as transações ativas de uma determinada conexão.

MON$STATEMENTS: Esta tabela está associada às instruções que estão sendo executadas em uma determinada base de dados. Algumas informações fornecidas por esta tabela estão quais instruções estão sendo executadas, como está o status desta instrução (em espera ou ativa), qual é a data e a hora de início destas instruções, qual a transação associada, e a conexão associada a esta instrução SQL que está sendo executada em um determinado momento.

Existem outras tabelas, no entanto essas são as tabelas mais importantes em se tratando em tabelas de monitoramento. É interessante avaliar, que essas tabelas se relacionam entre si, e pode-se fazer joins entre si para trazer os dados que são pertinentes para cada momento. Desta forma, as tabelas de monitoramento são diferentes da Trace API, mas podem ser um complemento quando precisa-se avaliar os dados que estão acontecendo em uma determinada base de dados.

no arquivo fbtrace.conf foi feita a configuração para definir quais as opções que serão auditadas nessa sessão. Para conseguir conectar no banco de dados utilizou-se a ferramenta IBExpert.

Pode-se observar o retorno do comando quando se conecta em uma determinada base de dados. No exemplo anterior o comando mostra três blocos que são respectivamente os retornos do Trace API. No primeiro e segundo bloco, observa-se que houve um evento trace_init. Este evento foi gerado porque foi iniciado o Trace API. No terceiro bloco houve outro evento que foi chamado de attach_database. Este é o

Seção Arquivo	Parâmetro	Descrição	Valor
Database	Enabled	Habilita ou desabilita o Trace.	True ou false
	Log_filename	Caminho e arquivo onde será guardado o log.	String
	Max_log_size	Tamanho máximo do arquivo de log. O sistema de log é rotacional.	Inteiro
	Include_filter	Audita Somente expressões SQL.	String (Expressão regular)
	Exclude_filter	Exclui as expressões SQL que não deseja visualizar.	String (expressão regular)
	Log_connection	Habilita ou desabilita o Log.	True ou False
	Connection_id	Habilita o Trace para uma ligação definida (Usa-se o id para explicitar qual ligação deseja). No caso do 0 todas.	Inteiro
	Log_transaction	Habilita o Log das transações (Start / Commit e Rollback).	True ou False
	Log_statement_prepare	Habilita os logs da preparação dos comandos.	True ou False
	Log_statement_free	Habilita os logs quando a transação é limpa.	True ou False
	Log_statement_start	Habilita os logs quando a transação é iniciada.	True ou False
	Log_statement_finish	Habilita os logs quando a transação é finalizada.	True ou false
	Log_procedure_start	Habilita o Log quando uma stored procedure é iniciada.	True ou false
	Log_procedure_finish	Habilita o Log quando uma stored procedure é finalizada.	True ou False
	Log_trigger_start	Habilita o Log quando uma stored procedure é iniciada.	True ou false
	Log_trigger_finish	Habilita o Log quando uma stored procedure é finalizada.	True ou False
	Log_context	Captura os valores que estão na variável RDB$Context.	True ou False
	Print_plan	Habilita a impressão qual o plano que foi utilizado.	True ou False
	Print_perf	Habilita a impressão detalhada dos dados de desempenho estatística.	True ou False
	Log_blr_request	Habilita log das requisições BLR.	True ou False
	Print_blr	Habilita a impressão do log BLR.	True ou False
	Log_dyn_request	Habilita o log das requisições DYN.	True ou False
	Print_dyn	Habilita a impressão das requisições DYN.	True ou False
	Max_sql_length	Define o tamanho máximo do sql a ser impresso, o restante é truncado.	Inteiro (Em caracteres)
	max_blr_length	Define o tamanho máximo da requisição BLR a ser impressa, o restante é truncado.	Inteiro (Em caracteres)
	max_dyn_length	Define o tamanho máximo da requisição DYN a ser impressa, o restante é truncado.	Inteiro (Em caracteres)
	max_arg_length	Tamanho máximo dos argumentos a serem impressos.	Inteiro (Em caracteres)
	max_arg_count	Numero máximo de impressões dos argumentos definidos.	Inteiro
Services	Enabled	Ativa ou não ativa o log dos serviços.	True ou False
	log_filename	Caminho e nome do arquivo aonde será feita a auditoria.	String
	max_log_size	Tamanho máximo do Log do arquivo definido em log_filename	Integer
	include_filter	Inclui ou não os serviços que serão feitos o Trace.	String (expressão regular)
	exclude_filter	Explicita quais os comandos serão excluídos.	String(expressão regular)
	log_services	Gerencia os eventos que serão mostrados.	True ou False
	log_service_query	Imprime o resultado dos serviços logados.	True ou False

Tabela 1. Configurações do arquivo FBTrace.conf

evento que é gerado quando uma nova conexão é feita na base de dados. Pode-se visualizar que além do tempo em que esta operação aconteceu, foi mostrada qual foi a base de dados que teve uma solicitação de conexão. Outras informações que são pertinentes: o usuário que fez a requisição para esta base de dados, qual é o protocolo que está sendo usado para fazer a conexão com a base, e qual foi o servidor que fez a requisição para a base de dados. Na linha abaixo ele mostra qual é a aplicação que fez a requisição para o servidor e qual a porta que a requisição está usando.

São várias as informações que se obtém utilizando os recursos da Trace API, uma vez que este foi só um exemplo e que foi avaliado somente as operações de conexão na base de dados. Além do mais existem vários tipos de auditoria.

A aplicação

A ideia da aplicação exemplo será desenvolver algumas rotinas para que quando o usuário selecione o que ele deseja auditar o programa faça a geração automática do arquivo de configuração e inicie o monitoramento da base de dados sem que o usuário tenha que ficar editando arquivos. Depois de escolhidas as opções o usuário irá clicar em um botão e o sistema irá abrir uma janela de prompt de comando e ficará aguardando os eventos serem lançados automaticamente na base de dados conforme outros aplicativos vão fazendo as requisições. Se acontecer alguma ocorrência que o usuário configurou, este evento será exibido no prompt de comando. No exemplo desta aplicação os dados serão tratados apenas em modo texto, mas nada impede de tratar esses resultados e gerar gráficos para verificar como estão os eventos e quais os comandos que são mais executados na base de dados, podendo desta forma facilitar o entendimento da auditoria.

Outro ponto que se pode ressaltar é filtrar no arquivo de configuração qual é a base de dados que se deseja fazer a auditoria, já que isso é possível e o arquivo de configuração permite que seja configurado especificando somente algumas bases de dados que estão locadas no servidor. Isso é outra possibilidade que pode ser explorada e incrementada para melhorar a auditoria dos dados de um determinado servidor Firebird.

Na **Figura 3** é exibida a aplicação desenvolvida para exemplo. Os componentes usados são basicamente: caixa de checagem para habilitar as opções desejadas, caixas de textos, botões para execução dos comandos, rótulos e um OpenDialog para selecionar o caminho do comando que faz a interface, o fbtracemgr.exe.

O usuário irá selecionar o comando que é responsável pela auditoria o fbtraceMgr e configurar o IP de onde há um servidor Firebird rodando, a porta que o servidor está rodando, o usuário e a senha deste servidor. Após a configuração destes procedimentos, pode-se iniciar o preenchimento do que se deseja auditar. Uma vez configurado, o usuário clica no botão Start Trace o programa abrirá uma tela como na **Figura 4** e começa a fazer a auditoria do que se foi selecionado. A figura exibe no resultado alguns eventos referentes a transações de dados que foram iniciadas e "comitadas" diversas vezes. Além dessas informações consegue-se observar qual é a base de dados que ocorreram estes eventos o usuário e também o aplicativo que fez essa solicitação juntamente com a sua porta. Nas últimas informações temos dados da transação, qual o tipo de isolamento se a transação aguardaria outra transação terminar em caso de seleção

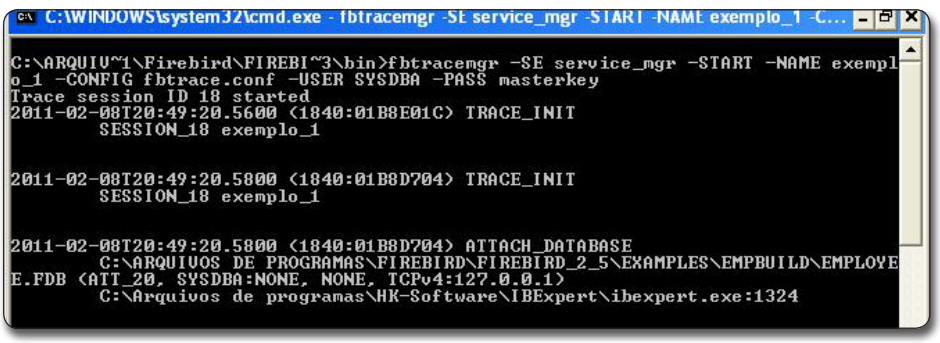


Figura 2. Resultado do comando FbTraceMgr

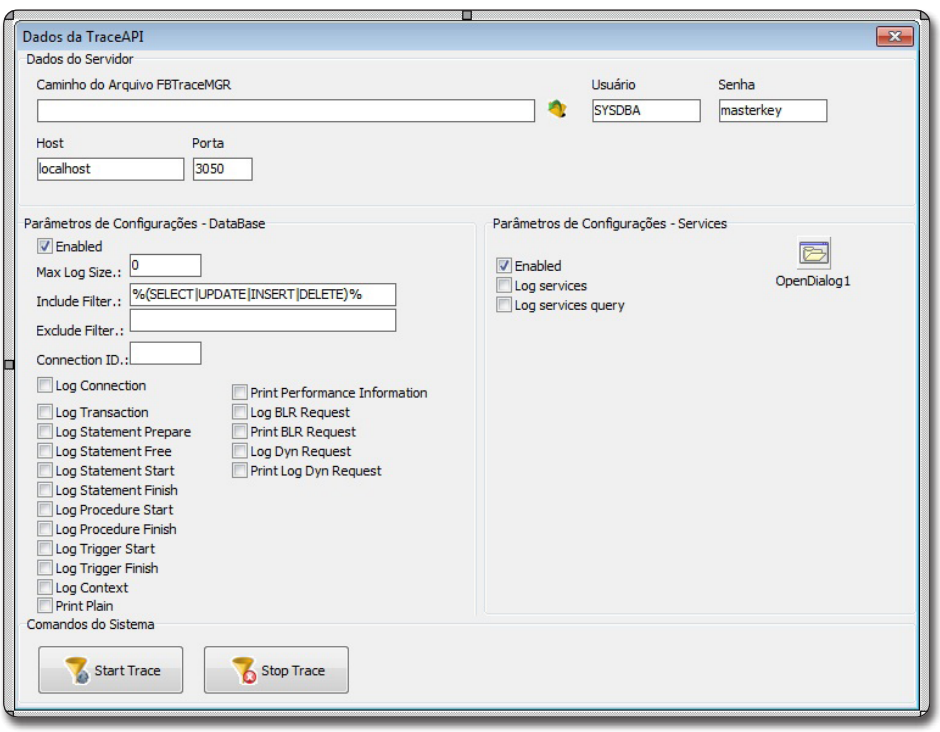


Figura 3. A aplicação de exemplo

ou manipulação dos mesmos dados e o tipo de acesso, se somente leitura ou se leitura e escrita.

Na **Figura 5** consegue-se observar o retorno de instruções SELECT. O interessante que se pode ressaltar é que nas instruções de DML (Data Manipulation Language ou Linguagem de manipulação de dados) consegue-se verificar quais são os parâmetros utilizados. Por exemplo, se é feito um select utilizando na cláusula where alguns parâmetros, consegue-se visualizar esses parâmetros no Trace API do servidor. Desta forma é possível, utilizando outras ferramentas em conjunto com a Trace API, verificar qual dos comandos DML estão impactando em uma queda de performance no banco de dados, ou verificar qual o comando que está travando alguma transação. São diversas as possibilidades que podem auxiliar o desenvolvedor.

O comando SQL que foi feito foi um select * from job where jog.country = 'USA' como pode observar no Trace API, consegue-se observar o parâmetro que foi utilizado, facilitando assim a auditoria.

Cada tipo de evento reporta um resultado diferente, de acordo com o esperado em cada campo, por exemplo, na transação se têm informações como o tipo de isolamento o tipo de acesso, etc. Na conexão com um database já não se tem essas informações, então, caso for fazer algo para tratar essas strings de retorno, tem que se atentar para este fato.

Entendendo a programação

Neste aplicativo foi utilizado alguns recursos da API do Windows para conseguir fazer a chamada a determinados programas externos ao Delphi, como por exemplo, o aplicativo fbtracemgr.exe. Neste exemplo foi aplicada a ShellAPI que faz chamadas a programas externos utilizando os recursos do Windows.

Na ShellAPI há um método que chama-se ShellExecute. Este método recebe alguns parâmetros e então ele executa o que é para ser executado. No código a seguir está a chamada do método, passando os parâmetros necessários para a chamada do fbtracemgr:

```
executavel := '"' + edit1.Text + '"';
ShellExecute (0,nil,  PWideChar(Executavel), PWideChar(' -SE ' + trim(edit2.
text)+'/' + EDIT3.TEXT + ':service_mgr -START -NAME traceLog -config ' + '"' +
ArquivoConfiguracao + '"' + ' -USER ' + Edit5.Text + ' -password ' + Edit6.text  ),'',
SW_MAXIMIZE);
```

Observa-se que o ShellExecute precisa de alguns parâmetros para poder funcionar. O primeiro parâmetro é o Handle da janela que no caso do exemplo é passado zero, o segundo parâmetro é o tipo de operação que o ShellExecute está esperando, foi passado nil, o terceiro parâmetro é o comando que deverá ser executado, fbtracemgr, já que o programa de exemplo aqui faz a interface para este comando. O quarto parâmetro são as opções que o comando fbtracemgr vai receber quando for chamado pela

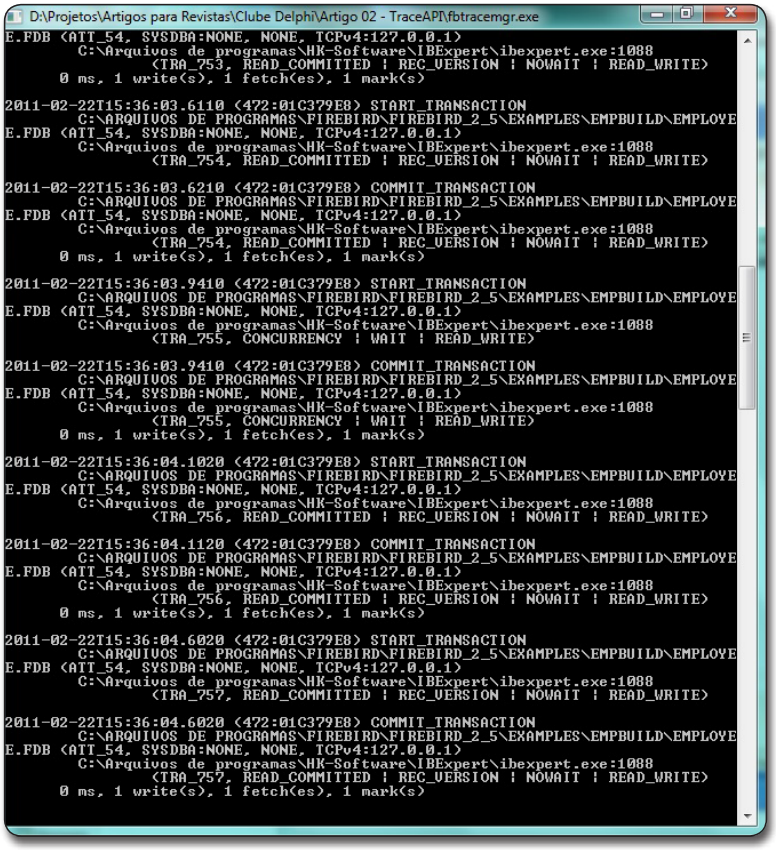


Figura 4. Resultado do monitoramento do Trace API

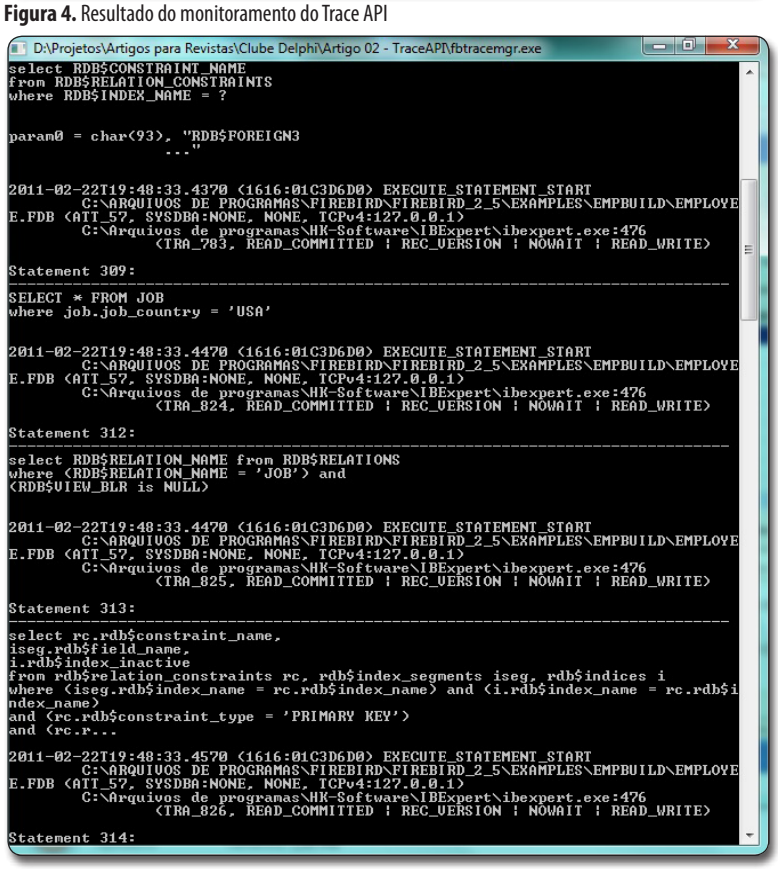


Figura 5. Resultado dos SQLs

Listagem 1. Exemplo parcial da montagem do arquivo de configuração

```
var
 ArquivoConfiguracao: string;
 i: Integer;
 Executavel: string ;
 ArqConfiguracao: TextFile;
begin
  ShowMessage('Para sair do TraceMode precione CTRL + C' + chr(13) +
        'ou utilize o botão de Stop Trace informando o id do trace');
  // início do trecho de configuração da Trace API
  i := length(edit1.Text);
  ArquivoConfiguracao := Edit1.Text ;
  while ArquivoConfiguracao[i] <> '\' do
    dec(I);
  delete(arquivoconfiguracao,i,(length(arquivoConfiguracao)+1)-i);
  ArquivoConfiguracao := ArquivoConfiguracao + '\fbtrace.conf' ;
  AssignFile(ArqConfiguracao, arquivoconfiguracao);
  Rewrite(arqConfiguracao);
  executavel := '"' + edit1.Text + '"' ;
  Writeln(arqConfiguracao, '<database>') ;
  Writeln(arqConfiguracao, '#Trace database events');
  if chebox_Enabled.Checked then
    Writeln(arqConfiguracao, 'enabled true')
  else
    Writeln(arqConfiguracao, 'enabled false');
  Writeln(arqConfiguracao);
  Writeln(arqconfiguracao, '# Trace on connection id');
  if length(Trim(EdiIdConnection.Text)) > 0 then
    Writeln(arqConfiguracao, 'connection_id ' + EdiIdConnection.Text)
  else
    Writeln(ArqConfiguracao, 'connection_id 0');
  Writeln(arqConfiguracao);
  Writeln(arqConfiguracao, '#SQL Queries include filter');
  if Length(Trim(EdiIncludeFilter.Text)) > 0 then
    Writeln(arqConfiguracao, 'include_filter ' + ediIncludeFilter.Text)
  else
    Writeln(arqConfiguracao, '#include_filter');
  Writeln(arqConfiguracao);
  Writeln(arqConfiguracao, '#SQL Queries Exclude filter');
  if Length(Trim(EdiExcludeFilter.Text)) > 0 then
    Writeln(arqConfiguracao, 'exclude_filter ' + EdiExcludeFilter.Text)
  else
    Writeln(arqConfiguracao, '#exclude_filter');
  Writeln(arqConfiguracao);
end;
```

Listagem 2. Comando para executar a parada do FBTraceMgr

```
var
  executavel: string ;
  Dados: string ;
begin
  executavel := '"' + edit1.Text + '"' ;
  if not InputQuery('Finalizar uma traceAPI',
    'Digite o ID da trace a ser finalizada', dados) then
    abort;
  ShellExecute(0, nil, pwidechar(executavel), PWideChar(' -SE ' +
    trim(edit2.text)+ '/' + EDIT3.TEXT + ':service_mgr -STOP -ID ' +
    dados + ' -USER ' + Edit5.Text + ' -password ' +
    Edit6.text),'', sw_maximize );
end;
```

API. O conteúdo do Edit2.text é o host (ou ip) da máquina aonde está rodando o Firebird, no Edit3.text está a porta que o servidor Firebird está rodando. Logo em seguida está a API do Firebird que é a service_mgr, a opção -start (aonde inicia a auditoria), a opção –Name que define o nome desta seção de auditoria, a opção –config que define o caminho do arquivo de configuração, que a aplicação montou, o usuário e a senha que estão nos respectivos Edit5 e Edit6.

O próximo parâmetro do ShellExecute é o diretório que está alocado o aplicativo que deseja iniciar. Neste caso não foi utilizada esta opção porque já foi passado o caminho completo juntamente com o nome do programa, e a última opção é o tipo de visualização que é desejada, no caso do exemplo, foi desejado que se mostrasse em tela maximizada, já que foi colocado o parâmetro sw_maximize.

Na **Listagem 1** está o exemplo da montagem do arquivo que é usado para passar como parâmetro na chamada da fbtracemgr. É mostrado como foi feito para gerar o arquivo de configuração que é utilizado para passagem de parâmetro para o comando fbtracemgr. Observa-se que de acordo com o que é selecionado nas caixas de seleção o arquivo é gerado com as opções desejadas. Então dessa forma é interessante que se parametrize o que é desejado ver na auditoria do banco. Como exemplo pode-se visualizar os inícios e os finais das transações, também podemos observar as conexões que estão sendo feitas na base de dados, os prepare statements de comandos e diversas outras informações e combinações que o arquivo nos permitir.

No botão de STOP TRACE, executa-se novamente o comando FBTracemgr para fazer a parada do servidor de auditoria. Na **Listagem 2** é mostrado como fazer a parada da auditoria caso ela já esteja rodando. Foi executado o comando ShellApi com alguns parâmetros diferentes. No caso a principal diferença é a troca do –START para –STOP e o acréscimo do parâmetro –ID que irá definir qual é o número do ID que será parado no processo de auditoria. Uma vez executado este comando o processo é automaticamente finalizado e esse ID é perdido não tendo mais como recuperá-lo (salvo na exceção de o servidor ser finalizado e iniciado novamente). Outra opção para finalizar o processo de auditoria é utilizando na própria tela da auditoria a opção CTRL + C.

Pode-se implementar também outros recursos que o fbtracemgr fornece que é o caso da suspensão e do novo início da auditoria. É bem interessante quando deseja auditar uma parte de algum sistema, ou seja, caso tenha que se executar todo um processo e a parte que interessa seja o meio da execução, pode-se iniciar a sessão e já suspendê-la até chegar o ponto que precisa ser observado.

Conclusão

Conforme o Firebird vai evoluindo, cada vez mais são disponibilizadas ferramentas para facilitar a vida dos desenvolvedores e das pessoas que o utilizam em meio de produção. No Firebird 2.1 foram acrescentadas as tabelas de monitoramento, agora

com o Firebird 2.5 a inclusão da Trace API. Podem-se esperar cada vez mais facilidades de monitoramento dessa base de dados. Por mais que o Firebird não necessite diretamente de um DBA, essas ferramentas podem identificar pontos onde é necessário melhorar a performance dos dados.

Além do mais algumas ferramentas de terceiros já estão no mercado e fazem uso desta tecnologia, facilitando a vida. É bom conhecer como funcionam as coisas para conhecer a tecnologia envolvida. Abraço a todos.

Jederson Zuchi
jederson@sygnux.com.br
www.twitter.com/jedersonzuchi
É diretor de sistemas industriais da Sygnux Software (www.sygnux.com.br) localizada em Monte Alto/SP. Formado pela Fatec/TQ, desenvolve com Delphi desda versão 3. Atualmente trabalhando com Delphi, Java e Plataforma .NET em diversos ERPs industriais. Professor do SENAC/SP e da Fatec/TQ lecionando em diversas matérias de desenvolvimento de software.

Página Oficial do Firebird.
www.firebirdsql.org

Página com várias documentações do IbPhoenix
http://www.ibphoenix.com/

Dê seu feedback sobre esta edição!

A Clubedelphi tem que ser feita ao seu gosto. Para isso, precisamos saber o que você, leitor, acha da revista!

Dê seu voto sobre este artigo, através do link:

www.devmedia.com.br/clubedelphi/feedback

Delphi Prism

Melhorias da linguagem – Parte

O Delphi Prism apresenta uma nova forma de indicar a definição de variáveis, além da forma tradicional utilizada em outras versões do Delphi, ou seja, em uma seção separada no início da definição de algum método. No Prism, isso não é mais necessário, podendo as variáveis serem criadas e inicializadas no formato inline (na linha de código onde ocorrerá a sua utilização ou existir a necessidade de definição da variável). A **Listagem 1** indica um exemplo desta utilização. Veja que o ArrayList é declarado da forma tradicional, já a variável "a" é declarada no interior do bloco de código.

Outra utilização de var é demonstrada no código da **Listagem 2** onde o tipo da variável é definido através de seu contexto de uso, sem a necessidade de indicar o tipo que será utilizado. A definição de variavel_inteira deduz o tipo de retorno da função retorna_inteiro, dessa forma utilizando seu tipo, neste caso um Int32. Além disso, a variável mensagem também utiliza o mesmo formato de dedução do tipo a ser utilizado, demonstrando que para vários tipos de dados é possível usar este modo de definição de variáveis. Este recurso é conhecido como Inferência de Tipos, ou Type Inference.

Utilizar var ainda vai além, permitindo usá-la na definição de métodos conforme pode ser visto no código da **Listagem 3**, onde a palavra var é utilizada nos parâmetros de um método. Observe o uso da palavra var na definição do método teste_de_metodo2, conforme descrito an-

Nota

A omissão do tipo não tem nada a ver com as variants do Delphi Win32, que perdem desempenho por não terem o tipo identificado na compilação, mas em runtime.

Resumo DevMan

De que trata o artigo:

O artigo demonstra algumas das principais melhorias e palavras reservadas da linguagem do Delphi Prism. Neste artigo, abordaremos preferencialmente recursos que possuem significativas melhorias / incrementos, ou mesmo que não existem, se comparados aos recursos da linguagem Delphi Win32 tradicional.

Para que serve:

As melhorias na linguagem servem para tornar a codificação mais enxuta e simples, ao mesmo tempo em que adicionam recursos poderosos presentes em linguagens orientadas a objetos mais modernas, como Java e C#.

Em que situação o tema é útil:

O tema pode ser utilizado para aumentar os conhecimentos e melhorar a produtividade utilizando as novidades e melhores funcionalidades da linguagem. Com isso tem-se um código mais refinado, elegante e eficiente. É muito útil para reduzir a codificação, visto que no Prism inúmeras construções da linguagem exigiram várias linhas de códigos adicionais se fossem escritas em Delphi Win32, para se obter o mesmo resultado.

Delphi Prism:

Durante o desenvolvimento do artigo, será visto de forma prática a utilização de palavras reservadas, como uso da palavra Var como uma nova forma de representar a definição de variáveis, o surgimento da palavra Yield como a maneira de possibilitar sair de uma iteração, armazenando o seu estado no momento da saída. Também é mostrada a liberação ou eliminação de recursos após o seu uso em determinados blocos de código através da estrutura Using. O uso de Require como alternativa para pré-condições de validação em métodos, onde o fluxo de execução do código para o método somente será executado se as regras definidas na utilização de Require forem satisfeitas. A substituição da unit pelo namespace onde estão presentes os tipos básicos que podem ser utilizados em aplicativos. A novidade do operador New, possibilitando criar novas instâncias de objetos em contraste com a forma tradicional utilizada em outras versões anteriores do Delphi, entre muitas outras novidades.

Listagem 1. Código para demonstrar o uso de var

```
method MainForm.button1_Click
 (sender: System.Object; e: System.EventArgs);
var
  x: ArrayList;
begin
  x := new ArrayList();
  var a : String;
  a := "teste";
  x.Add(a);
end;
```

Listagem 2. Código utilizando var com um tipo de dados inferido pelo contexto

```
function MainForm.retorna_inteiro: integer;
begin
  result := 111;
end;
method MainForm.teste_de_metodo;
begin
  var variavel_inteira := retorna_inteiro();
  variavel_inteira := variavel_inteira + 1;
  var mensagem := 'Olá mundo!';
end;
```

Rodrigo Araujo

teriormente. Essa utilização e o uso da inicialização da variável anterior como 'teste' permite realizar a chamada do método de formas distintas, tais formas podem ser visualizadas na **Listagem 4**.

Neste código é possível verificar formas diferentes de chamada do método. No primeiro caso, sem a passagem de parâmetros, a variável "a" será inicializada com a palavra "teste" (veja o uso de parâmetros padrão), e ao sair do método a variável "b" conterá o valor da variável "a". Na segunda chamada do método, a variável "a" conterá a string "devmedia" e, ao entrar no método, a variável "b" continuará a armazenar o valor definido na chamada anterior do método, no caso a string "teste", como se fosse um histórico do último valor atribuído a variável "a". Somente na saída do método ocorrerá a atribuição de um novo valor. Além disso, a linha de código comentada não poderá ser utilizada desta forma, devido à utilização de var na definição do método, pois ocorrerá um erro em tempo de compilação. Esta forma de utilização de var, presente e aprimorada no Prism, é semelhante ao Pascal tradicional onde temos variáveis passadas por valor ou referência.

Yield

O Yield é utilizado para sair de uma iteração armazenando o seu estado no momento da saída, ou seja, em uma estrutura de repetição, irá realizar repetições através dos objetos retornados pelo método utilizado, criando um mecanismo de estado onde será possível conservar este estado sem a necessidade de manutenção através de código.

Yield poderá trabalhar em conjunto com Iterators, uma vez que o código começará a enumerar a sequência de saída a cada iteração da estrutura de repetição utilizada e o Iterator será executado para fornecer os elementos desta sequência.

O código da **Listagem 5** demonstra a utilização de um Yield em conjunto com um Iterator para retornar uma sequência de inteiros que irão representar os números ímpares de zero a cem.

As duas primeiras linhas indicam a definição do método como sendo uma sequência de inteiros e a utilização do Iterator, dessa forma, o compilador através da diretiva de Iterator saberá que o método não é comum e receberá um tratamento diferenciado devido à forma de iteração que será utilizada com Yield. Em seguida, ocorre a implementação do método, utilizando uma estrutura de repetição e a palavra Yield. A cada chamada de Yield um valor será armazenado na sequência e retornado para o método GetNumImpares. Após armazenar o valor, a estrutura de repetição seguirá a sua execução indo para o próximo passo de execução, e assim por diante, até chegar ao seu término.

Para utilizar os resultados obtidos, o código da **Listagem 6** demonstra como conferir os dados armazenados no método em questão. No código é utilizado o método GetNumImpares e seus resultados são exibidos através de mensagem em tela para o usuário, utilizando um MessageBox. A cada iteração da estrutura for each é apresentada uma mensagem com um valor armazenado na sequência de inteiros. Yield também pode ser utilizado com IEnumerable a fim de facilitar e auxiliar na programação, assim, Yield realizará todo o trabalho de armazenamento e retorno das informações pretendidas.

Como comparativo para verificar a diferença de utilização de Yield, o código da **Listagem 7** demonstra o uso de uma forma tradicional de programação sem

Listagem 3. Código utilizando var na definição de um método

```
method MainForm.teste_de_metodo2(
          a : String := 'teste'; var b : String);
begin
 b := a;
end;
```

Listagem 4. Código realizando a chamada de um método

```
method MainForm.teste_de_metodo;
begin
 teste_de_metodo2();
 teste_de_metodo2('devmedia');
 //teste_de_metodo2('devmedia', 'delphi');
  comentado, pois ocorrerá erro devido ao uso de var
end;
```

Listagem 5. Código utilizando Yield

```
private
 method GetNumImpares: sequence of Integer;
iterator;

method MainForm.GetNumImpares: sequence
of Integer;
begin
 for i: Integer :=1 to 100 step 2 do
  yield i;
end;
```

Nota

Iterator é um tipo especial utilizado para retornar uma sequência, quando executado, o corpo do método do Iterator irá criar um objeto que representará esta sequência onde ele for utilizado.

o uso de Yield, já o código da **Listagem 8** irá auxiliar para indicar as diferenças. O cálculo das raízes quadradas de cada elemento da estrutura de repetição é armazenado em uma lista através da opção Add e, ao final, retornado como resultado. Existem algumas diferenças entre os códigos apresentados, usando maneiras distintas de se obter o mesmo resultado e exemplificando ainda melhor a utilização de Yield, que é mais enxuto. Além disso, na definição do método que utiliza Yield deverá conter a palavra Iterator, semelhante à **Listagem 9.**

Listagem 6. Utilização do método

```
var numeros := GetNumImpares();
for each n in numeros do
 MessageBox.Show(n.ToString());
```

Listagem 7. Utilização de um método sem o uso de Yield

```
method MainForm.Raiz_Quadrada: IEnumerable
 <Int32>;
begin
 var raizes : List<System.Int32> := new  List
 <System.Int32>();
 for i: Integer := 1 to 9 do
  raizes.Add(i * i);
 result := raizes;
end;
```

Listagem 8. Utilização de um método com o uso de Yield

```
method MainForm.Raiz_Quadrada2: IEnumerable
<Int32>;
begin
 for i: Integer := 1 to 9 do
  yield  (i * i);
end;
```

Listagem 9. Definição do método que irá utilizar Yield

```
method Raiz_Quadrada2: IEnumerable<Int32>; iterator;
```

Listagem 10. Uso de using em um bloco de código

```
using y := new ArrayList() do
begin
 y.Add('teste');
end;
//MessageBox.Show(y); //caso descomente, ocorrerá
erro indicando que y não está definido
```

Listagem 11. Alternativa de código sem o uso de using

```
with y := new ArrayList() do
try
 y.Add('teste');
finally
 if (y is IDisposable) then
  (y as IDisposable).Dispose();
end;
//MessageBox.Show(y); //caso descomente ocorrerá
erro indicando que y não está definido
```

Using

O using surge para auxiliar na liberação ou eliminação de recursos logo após o seu uso em determinadas combinações com blocos de código, funcionando de maneira semelhante a instruções try e finally. A **Listagem 10** ilustra o exemplo de sua utilização.

A utilização de using indica que a variável y criada estará presente e disponível somente dentro do bloco de código do using, dessa forma, se a linha que contém o MessageBox for descomentada, ocorrerá um erro, indicando que a variável y não está definida.

Using deverá conter uma construção que suporte o padrão .NET da interface IDisposable, possibilitando assim a chamada de Dispose para, enfim, liberar o recurso utilizado, caso isso não aconteça a chamada a Dispose será ignorada e, no momento corrente, o Garbage Collector não entrará em ação.

De maneira semelhante utilizando a interface IDisposable, mas sem o uso de using, como alternativa de código, a **Listagem 11** demonstra uma forma para melhor exemplificar a situação demonstrada no código anterior, caso fosse utilizado try finally para obter o mesmo resultado. Veja que é necessário realizar a chamada de Dispose para a variável y, que foi recentemente criada, liberando assim seus recursos. Ao utilizar using, esta preocupação não é necessária.

Nota

O IEnumerable serve para possibilitar iterações com um determinado conjunto de dados que irão representar valores, dessa forma, o compilador saberá se comunicar e auxiliará na utilização dessa estrutura representada.

Nota

O GC ou Garbage Collector é um processo utilizado para o gerenciamento de memória, com ele é possível recuperar áreas de memória que estavam sendo utilizadas. É o responsável por analisar endereços de memória não referenciados e liberá-los para que outros aplicativos possam utilizar, ou seja, otimizar a alocação de memória.

Require

Representa uma pré-condição que será avaliada e é necessário ser verdadeira para dar continuidade ao fluxo de execução de um código. Require é utilizada, principalmente, ao início de um método tendo por principais finalidades a verificação da validade de algum parâmetro de entrada, variáveis ou para determinados estados de objetos exigidos pelo método onde está sendo utilizado.

A **Listagem 12** demonstra a utilização de require em um determinado trecho de código. Antes da execução do código do método, é indicado o código que irá representar require, este código irá realizar uma validação na variável valor2, neste caso, a condição demonstrada não permitia que o valor seja zero, pois ocorrerá um erro de divisão. Dessa forma, caso o valor informado seja zero, require irá gerar uma exceção de código e uma mensagem será exibida na tela.

A **Figura 1** demonstra a mensagem que será indicada. É possível verificar a pré-condição e a indicação do que está sendo analisado (neste caso valor2<> 0), a informação das linhas onde a condição não foi validada pelo require e em qual arquivo está presente esta condição.

Namespace

O namespace substitui a palavra-chave unit no cabeçalho de um arquivo em Delphi. Um namespace, em .NET de forma simplificada, organiza os objetos ou funcionalidades de um assembly. Cada namespace contém tipos básicos que podem ser utilizados em aplicativos. O namespace tem por principal objetivo evitar a redundância de definições de nomes, além de tentar organizar e agrupar por tipos de objetos semelhantes ou com funcionalidades parecidas. Exemplo: O namespace definido por System.IO possui funcionalidades específicas para entrada e saída de dados.

Quando o Delphi Prism foi projetado, um dos objetivos era oferecer suporte a namespaces de maneira adequada para a programação. Dessa forma, interagindo com o modo como o CLR trata os namespaces, de forma semelhante ao utilizado

em outras linguagens de programação e permitindo a criação de bibliotecas com Delphi Prism que possam ser utilizadas de maneira intuitiva em outras linguagens como C# e Visual Basic.

O Delphi Prism apresenta alguns pontos que se destacam na utilização de namespaces e que levaram a sua eficaz utilização, tornando-o bastante intuitivo. Alguns destes são:

- Pode-se utilizar o mesmo nome de namespace para arquivos distintos e no mesmo projeto ou na utilização de DLL de terceiros. Esses arquivos serão tratados de maneira distinta como se fossem um namespace maior e todas as classes poderão ver e referenciar uns aos outros, além disso, referências circulares também serão permitidas;
- As referências circulares não causam problemas, o seu tratamento é dado de maneira hierárquica nos arquivos que estão sendo utilizados, respeitando as referências das classes;
- Independentemente do namespace que é definido para um determinado arquivo, sempre poderá ser declarado classes em qualquer outro namespace arbitrário, utilizando o nome completo na declaração da classe e nas implementações de métodos, possibilitando assim que existam classes de um único arquivo em namespaces distintos;
- Os tipos estarão todos disponíveis, aqueles que foram referenciados no Assembly e os tipos definidos no projeto atual, poderão ser utilizados através de seu nome completo, independente de seu namespace estar presente na cláusula uses, sendo possível somente utilizar o namespace que necessitar a referência ou os seus tipos presentes neste namespace.

Além disso, como um ótimo recurso, para o uso de grandes hierarquias aninhadas de namespaces, o Delphi Prism possibilita a utilização do caractere curinga * em sua cláusula de uses, para utilizar toda uma hierarquia presente em um namespace. Como exemplo, o uso de System.Xml.* permitirá o acesso a todo o conjunto de classes XML, incluindo os localizados em sub-namespaces como System.Xml.Path.

Na **Listagem 13** é visualizada a definição de um namespace e a utilização do caractere curinga *. De acordo com a definição anterior, o namespace receberá o nome de WindowsAppTestePrism, na cláusula uses a chamada a System.Xml.* e o uso do caractere "curinga" permitirão a utilização de todo o namespace presente em System.Xml, como exemplo, System.Xml.Xsl, System.Xml.Linq, System.Xml.Schema, System.Xml.Serialization entre outros.

New

O operador New permite criar em determinados momentos ou em algumas partes de código novas instâncias de objetos com o objetivo de utilizar as propriedades presentes nestes objetos. Seu funcionamento é semelhante à palavra Create utilizada no Delphi Win32 tradicional para representar a chamada do construtor.

Na **Listagem 14** é possível verificar a sua utilização. A maneira de criar a variável x é bastante intuitiva e a linha comentada demonstra a forma tradicional utilizada em outras versões do Delphi.

Além disso, o Delphi Prism apresenta suporte ao uso de parâmetros nomeados em uma chamada de construtor para a definição de valores de campo ou indicação de propriedades, permitindo que sejam definidos valores em uma propriedade sem ter que realizar a declaração de variáveis. A definição da classe Pessoa (**Listagem 15**) indica que ela possui uma propriedade Nome do tipo String que será utilizada como referência para exemplificar a situação mencionada anteriormente.

Na **Listagem 16** é possível verificar a utilização de New para a classe Pessoa e o uso direto da atribuição do Nome sem a necessidade de criar variáveis, dessa forma, utilizando parâmetros nomeados

Nota

A interface IDisposable é utilizada para liberar recursos não gerenciados, o Garbage Collector automaticamente libera a memória alocada para um objeto gerenciado quando este objeto não estiver mais sendo utilizado, mas não é possível prever quando essa liberação irá ser realizada. Além disso o GC não possui conhecimento de recursos não gerenciados como identificadores de janelas (Handles), arquivos abertos diretamente no SO etc.

Listagem 12. O uso de require em um método

```
method MainForm.calcula(valor1 : Int32;
 valor2: Int32);
require
 valor2 <> 0;
begin
  valor1 := valor1 / valor2;
end;
```

Listagem 13. Definição de um namespace em um arquivo de código

```
namespace WindowsAppTestePrism;

interface

uses
 System.Drawing,
 System.Collections,
 System.Collections.Generic,
 System.Windows.Forms,
 System.Xml.*,
 System.ComponentModel;
```

Figura 1. Mensagem indicada na tela ao validar um código através de uma condição

Nota

Assembly é um conjunto de tipos e recursos capazes de se agrupar e interagir para formar funcionalidades de bibliotecas distintas. Além disso, o Assembly contém toda a especificação de como os elementos se relacionam e quais as suas dependências entre si.

Nota

O CLR (Commom Language Runtime) é a "máquina virtual" do .NET. É nele que ocorre a chamada ao compilador JIT (Just In Time, ou tempo de execução) que é o responsável por traduzir ou codificar o código IL em um código nativo que o computador possa executar.

da própria classe. Este recurso também é conhecido como construtor estendido ou ainda object initializer.

Finalizer

Fornece recursos para limpar as classes que não foram devidamente eliminadas. O Finalizer consome processamento para o Garbage Collector, por isso sua utilização deverá ser calmamente planejada, e normalmente é utilizado quando houver bastante necessidade através de uma implementação declarativa no código fonte utilizado.

Nota

Quando notify é utilizado em propriedades, a classe irá implicitamente utilizar um System.ComponentModel.INotifyPropertyChanged e uma interface System.ComponentModel.INotifyPropertyChanging, e isso irá ocorrer assim que um evento realizar alguma modificação de valor. INotifyPropertyChanged é utilizada para notificar os clientes que o valor de uma determinada propriedade foi alterado. Já a INotifyPropertyChanging é usada para indicar que o valor está sendo modificado.

Na **Listagem 17** é visualizado o uso de Finalizer na definição da classe Cadastro. Ocorreu a definição de uma classe Cadastro e o uso de Finalizer para realizar implementações de finalização e eliminação de recursos que não são mais necessários. Vale notar que Finalizer sempre será protected/override não importando onde esteja localizada a sua declaração.

Outro uso de Finalizer ocorre na implementação de IDisposable seguindo um padrão para garantir que os recursos não sejam esquecidos pelo caminho ou que sejam lançados mais de uma vez. Este uso é observado principalmente em classes que gerenciam algum estado de recursos como manutenção de arquivos, identificadores de banco de dados, streams ou instâncias de outras classes.

A **Listagem 18** apresenta a definição da classe pessoa em conjunto com IDisposable. Existe o uso de IDisposable, a definição de uma variável booleana para indicar se a classe já liberou seus recursos não mais utilizados, o método Dispose e a declaração Finalizer em conjunto com Finalize, que é opcional.

Nos códigos da **Listagem 19** é possível verificar a definição de Finalize, a liberação de recursos e a interessante chamada a GC.SuppressFinalize que irá indicar ao sistema que o objeto utilizado não deverá receber chamadas de Finalizer, pois esta tarefa, devido à implementação descrita, será desnecessária.

Os recursos utilizados pelos objetos gerenciados serão valorizados pelo Garbage Collector e qualquer classe que crie recursos não gerenciados, como arquivos ou sockets poderá conter um Finalizer para liberá-los e normalmente isso poderá ocorrer antes de terminar a execução do aplicativo.

O uso de Finalizer deverá ser observado com bastante cuidado e somente ser utilizado quando for necessário, visto que a principal penalidade é o desempenho ao que está associado, pois o Garbage Collector irá realizar no mínimo duas passagens: a

Listagem 14. Utilização de new para criar novos objetos a serem utilizados

```
x := new ArrayList();
...//x := TStringList.Create; // código utilizado em
Delphi win32 tradicional
```

Listagem 15. Criando uma classe para utilização de new

```
type
  Pessoa = class
  public
    constructor; empty;
    property Nome: string;
  end;
```

Listagem 16. Utilizando new com parâmetros nomeados

```
var x := new List<Pessoa>;
x.Add(new Pessoa(Nome:='Rodrigo'));
MessageBox.Show(x[0].Nome.ToString());
```

Listagem 17. Uso de finalizer

```
type
  Cadastro = class
  public
    constructor; empty;
    finalizer;
  end;
implementation
finalizer Cadastro;
begin
  //implementações a serem definidas...
end;
```

Listagem 18. Definição de classe com IDisposable

```
type
  pessoa = class(IDisposable)
  private
    disposed: Boolean;
    method Dispose(disposing: Boolean);
  public
    method Dispose;
    finalizer Finalize;
  end;
```

Listagem 19. Implementação dos métodos e uso do Garbage Collector

```
method pessoa.Dispose;
begin
  Dispose(true);
  GC.SuppressFinalize(self);
end;
method pessoa.Dispose(disposing: Boolean);
begin
  if not disposed then
  begin
    if disposing then
    begin
      // liberação de recursos
    end;
    disposed := true;
  end;
end;

finalizer pessoa.Finalize;
begin
  Dispose(false);
end;
```

Listagem 20. Uso de empty para a definição de um método

```
method ConsultaDados; virtual; empty;
```

Listagem 21. Demonstração do uso de notify em propriedades de uma classe

```
type
  MinhaClasse = class
  public
    property Valor: Integer; notify;
    property Nome: string;
    notify 'Alteração de Nome';
  end;
```

Listagem 22. Implementação do uso de notify

```
method MainForm.button1_Click(sender:
System.Object; e: System.EventArgs);
begin
      var  v:= new MinhaClasse;
      v.PropertyChanged +=
    method (sender: Object; args:
    PropertyChangedEventArgs)
   begin
     MessageBox.Show(String.Format(
      'Algum valor modificado, propriedade  {0}',
      a1rgs.PropertyName));
   end;
  v.nome := 'Rodrigo';
end;
```

passagem forçada com o Finalizer e a passagem geral de limpeza, se existirem muitas instâncias de objetos criados, o Finalizer irá realizar chamada para cada uma delas o que poderá sobrecarregar a aplicação.

Empty

A diretiva Empty pode ser utilizada em declarações de métodos para indicar que nenhum método de execução será apresentado. Em contraste com métodos abstratos, o compilador irá gerar um esqueleto vazio para o método e o mesmo poderá até ser chamado. Suas principais utilidades implicam na adição de métodos nas classes base que podem ser substituídas nas classes descendentes e não ocorrendo de forma obrigatória. Além disso, é útil na definição de interfaces de classes que mais tarde serão desenvolvidas e implementadas.

A **Listagem 20** indica o uso de Empty na definição de um método chamado ConsultaDados. Este método não possui implementação de código, apenas esta definição com a utilização de Empty, possibilitando sua posterior implementação quando existir necessidade. Para os parâmetros de saída e resultados serão devolvidos valores nulos e para os demais valores de tipos será atribuído valor zero ou false.

Notify

É utilizado principalmente para a notificação de propriedades, normalmente sendo apresentada como uma característica da linguagem que permite realizar notificações para uma propriedade quando um valor desta propriedade é modificado ou adicionado.

O código da **Listagem 21** apresenta a definição de uma classe utilizando Notify em suas propriedades. A classe definida chama-se MinhaClasse e apresenta duas propriedades: Valor e Nome, ambas com o uso de Notify, a diferença está presente apenas na propriedade Nome, pois existe uma mensagem definida por 'Alteração de Nome'. Esta mensagem será apresentada na execução do código da **Listagem 22** que define uma forma de utilização do Notify definido anteriormente.

A atribuição de código à PropertyChanged indicará que ao ocorrer alguma alteração nas propriedades descritas por Notify, o evento será executado e uma mensagem será apresentada ao usuário. Neste caso à propriedade Nome foi atribuído um valor e a mensagem indicará o PropertyName definido na propriedade, representado por "Alteração de Nome". Caso fosse utilizado e atribuído um valor a propriedade Valor, PropertyName irá indicar como mensagem Valor que é o nome definido para a propriedade e indicando qual foi a propriedade utilizada. Dessa forma sendo uma maneira bastante prática de utilizar Notify para situações em que haja necessidade de definição de valores arbitrários ou uma busca por quais propriedades foram alteradas.

Conclusão

A utilização do Delphi Prism possibilita novas e melhores maneiras de programação levando em conta as melhorias de linguagem existentes no Prism. Elas vieram para ampliar o conhecimento e aperfeiçoar o desenvolvimento de aplicações de maneira em geral. O bom uso e conhecimento dessas funcionalidades irão aumentar a produção e permitirão um tratamento mais eficaz ao desenvolver sistemas.

O artigo demonstrou a utilização de algumas das principais melhorias como o uso de var, a utilização de Yield, Using, Require, Namespace, New, Finalizer, Empty e Notify para realizar notificações em propriedades de uma classe, possibilitando, assim, refinamentos e aplicações de conhecimento para serem atribuídas no desenvolvimento.

As funcionalidades visualizadas no artigo poderão ser usadas no desenvolvimento, visando sempre uma melhor e mais enxuta forma de realizar a programação, logo, facilitando a vida de quem irá desenvolver os aplicativos e melhorar a sua produção.

Rodrigo Araujo
rodrigo.arsi@gmail.com
Bacharel em Ciência da Computação pela UFSM – Santa Maria/RS trabalha com Análise e Desenvolvimento de Software há mais de 10 anos. Desde 2003, utiliza tecnologias .NET.

Dê seu feedback sobre esta edição!

A Clubedelphi tem que ser feita ao seu gosto. Para isso, precisamos saber o que você, leitor, acha da revista!

Dê seu voto sobre este artigo, através do link:

www.devmedia.com.br/clubedelphi/feedback

DataSnap no Delphi X

Conectando um mesmo servidor de apli

Sistemas multicamadas, como o próprio nome diz, caracterizam a divisão em camadas de uma aplicação, onde cada camada é uma unidade lógica. Essas unidades funcionam em conjunto, porém em processos separados e em máquinas separadas, cada qual com uma especificação própria. Por estarem separadas fisicamente, as camadas precisam partilhar de uma comunicação entre si, que pode ser via rede local ou através da internet, onde são utilizados os protocolos de comunicação TCP/IP ou HTTP. Outra característica relevante é a centralização das regras de negócio em sua camada intermediária, o que facilita a atualização da aplicação, visto que, em uma eventual mudança das regras de negócio, somente uma camada seria atualizada. Em consequência disso, a camada de apresentação, que é representada pelas aplicações clientes, se abstém dos códigos de negócio, se tornando clientes leves, ou mais comumente chamados Thin Clients.

O Delphi 2009 marcou a mudança em termos de desenvolvimento multicamadas no Delphi, com a remodelagem de seu Framework DataSnap que o acompanha desde a versão 3. Dentre as mudanças realizadas está a eliminação de sua dependência com a tecnologia COM para comunicação entre a parte cliente e servidor e a adição de um novo driver dbExpress de conexão, exclusivo para DataSnap. Isso tornou muito claro e prático o desenvolvimento DataSnap no Delphi desde então. A mudança foi tamanha que na época batizaram a versão como DataSnap 2009, para que não fosse confundido com as versões anteriores da tecnologia. Já em termos de conectividade com outras plataformas, até então era possível somente construir clientes nativos .NET, utilizando o Delphi Prism 2009. Com o lançamento do Delphi XE, este leque foi aumentado. Agora é possível criar clientes nativos em PHP, JavaScript, C++ e .NET utilizando as ferramentas RadPHP, C++ Builder XE e Delphi Prism XE.

Portanto, essa será a abordagem do artigo, ou seja, construir uma solução multicamadas utilizando o Framework DataSnap do Delphi XE, onde será desenvolvido um servidor de aplicações tradicional que estará acessível para clientes distintos, nos ambientes Web e Desktop. Esses clientes serão os mais "leves" possíveis, tendo o mínimo de código possível, somente para poder consumir os métodos de negócio já disponíveis no servidor e manipulação de alguns controles em tela.

Resumo DevMan

De que trata o artigo:

Demonstrar de forma prática como é possível utilizar o Framework DataSnap do Delphi XE para a construção de uma solução multicamadas, cujo servidor de aplicação disponibilizará seus métodos a múltiplos clientes, de tecnologias heterogêneas Web e Desktop, tais como VCL Forms e VCL for Web, PHP e .NET.

Para que serve:

O conceito de multicamadas serve para dividir a aplicação em várias partes, cada qual com objetivos fortemente definidos. No modelo a ser abordado serão três: dados (banco de dados), regras de negócio (servidor de aplicação) e apresentação (aplicações clientes em variadas tecnologias Web e Desktop). O uso de vários tipos de interfaces clientes serve para que códigos sejam reaproveitados no servidor, não importando a forma como os dados forem apresentados (se na Web, Desktop ou ambos).

Em que situação o tema é útil:

A abordagem do artigo visa, de certa forma, incentivar os desenvolvedores que ainda não utilizam soluções multicamadas DataSnap a usarem e até mesmo migrarem suas aplicações Client/Server. Dessa forma, obtém-se uma maior flexibilidade, como por exemplo, a partir de um único servidor de aplicação central disponibilizar seus métodos de negócio para N clientes distintos. É útil para empresas que desejam oferecer mais de um tipo de interface para uma mesma solução servidora.

DataSnap: um servidor e vários tipos de interfaces clientes:

Neste artigo será construída uma solução multicamadas DataSnap acessível através de diferentes tecnologias, promovendo o desenvolvimento distribuído em camadas com interoperabilidade. Para valorizar ainda mais o exemplo, estas aplicações serão construídas utilizando além do próprio Delphi XE, outras ferramentas da própria Embarcadero como o RadPHP XE e o Delphi Prism XE, fazendo com que o desenvolvedor Delphi se sinta mais familiarizado.

O modelo 3 camadas

Quando se fala em multicamadas no Delphi logo se pode interpretar como sendo seu modelo mais simplista que é o modelo conhecido como "modelo 3 camadas", que será o abordado neste artigo. Conforme pode ser visto na **Figura 1**, as camadas deste modelo são:

• Dados: como a própria descrição sugere, é a camada onde estarão os dados da

E
cação a partir de clientes Web e Desktop

FABRÍCIO HISSAO KAWATA

aplicação, ou seja, o Sistema de Gerenciamento de Banco de Dados (SGBD) propriamente dito. Esta camada comumente está localizada fisicamente num servidor dedicado ou junto da camada de Regras de Negócio;

• Regras de Negócio: esta é a camada que faz acesso direto à camada de dados, bem como centraliza toda a regra de negócio da aplicação. Logo, sua função é prover os métodos de negócio e o serviço de dados à camada cliente. Em vista disso, seu acesso deverá estar disponível a todos os clientes. Reside, na maioria das vezes, em uma máquina servidora dedicada;

• Cliente: representa a camada de interface com o usuário. Em outras palavras, é a aplicação que fica na máquina do usuário. Neste modelo, caracteriza-se por ser leve, uma vez que todo o código de negócio já está concentrado no servidor e todo o processamento se dará também naquela camada.

Ainda observando a **Figura 1,** percebe-se a clara divisão de responsabilidades entre as camadas neste tipo de abordagem. Isso torna o código da aplicação mais limpo, pois, ao contrário do que comumente acontece em projetos Client/Server, em que há uma mistura de código de regras de negócio com o que trata da parte visual da aplicação, em multicamadas o desenvolvedor tende a separá-los.

Realmente compensa desenvolver neste modelo?

Visto algumas das vantagens do desenvolvimento multicamadas, ainda deve ter ficado alguma dúvida com relação a se realmente compensa adotar este tipo de solução, aparentemente mais complexo, para a construção de aplicações. Em vista disso, serão apresentadas e reforçadas mais algumas das vantagens do modelo multicamadas.

Centralização das regras de negócio

Como já mencionado anteriormente, as regras de negócio ficam centralizadas na camada intermediária, mais precisamente no servidor de aplicação. Portanto, quaisquer aplicações clientes que o projeto venha a ter, partilharão das mesmas regras. Numa situação real, caso uma regra de negócio da empresa venha sofrer alguma modificação, somente o servidor precisa ser atualizado, não sendo necessária a redistribuição de todas as aplicações clientes.

Clientes leves

Também chamados de Thin Clients, as aplicações clientes sofrem grande diminuição de complexidade e tamanho, principalmente com a redução de código, que é direcionado, em sua maioria, para controles de tela (Ex.: validações de campos), já que as regras de negócio já estão codificadas no servidor.

Economia

Ainda sobre clientes leves, como o processamento maior se dará no servidor, faz com que as estações clientes não necessitem de grandes investimentos em termos de recursos de hardware. Outra economia que se tem é com relação ao número de licenças de banco de dados, já que todas as aplicações clientes se conectam à camada intermediária e somente o servidor de aplicação necessita de uma conexão ativa com o banco.

Escalabilidade

No modelo multicamadas torna-se possível o trabalho com recursos de Failover e Load Balance. Como o servidor de aplicação irá tomar posse do maior processamento da solução, há de se levar em conta questões como redundância e balanceamento de carga. Imagine o caso onde o servidor falhe ou fique sobrecarregado de processamento, então se faz necessário

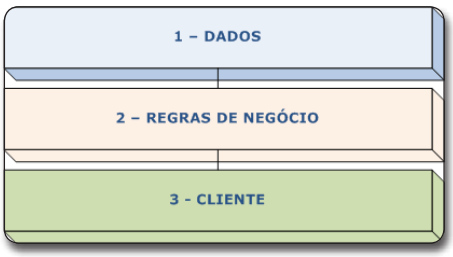


Figura 1. Modelo 3 camadas

Nota do DevMan

TCP/IP é o acrônimo de dois outros protocolos: TCP (Transmission Control Protocol - Protocolo de Controle de Transmissão) e IP (Internet Protocol - Protocolo de Interconexão). Pode ser definido como um conjunto de protocolos de comunicação, ou seja, envio e recebimento de dados, entre computadores em rede.

HTTP é a sigla para o termo em inglês Hipertext Markup Language. É o protocolo que realiza a transferência de dados entre computadores, utilizando mensagens para a comunicação entre as partes cliente e servidor.

COM (Component Object Model) é uma tecnologia da Microsoft que viabiliza a comunicação entre aplicações cliente e servidor, utilizando interfaces. Uma interface COM é a maneira como um objeto expõe sua funcionalidade ao meio externo.

Nota do DevMan

SGBD ou Sistema de Gerenciamento de Banco de Dados pode ser definido como um agrupamento de programas que tornam possível o armazenamento, modificação e extração de informações de um banco de dados. Alguns exemplos de SGBDs: Firebird, Oracle, MySQL, PostgreSQL, Sybase, Interbase etc.

Aplicações Client/Server são também conhecidas como "modelo 2 camadas". Neste modelo as regras de negócio podem ficar no servidor, na forma de objetos de banco de dados (como Stored Procedures ou Triggers), no Cliente, na forma de código de programação, ou em ambos. Além disso, todo o processamento envolvido nas atividades sistêmicas também pode ser dividido entre as camadas, variando de acordo com a estruturação da aplicação)

que outro esteja disponível para assumir ou dividir a carga.

Distribuição geográfica

As camadas, por estarem separadas fisicamente, podem estar distribuídas geograficamente. Um caso clássico que pode ser tomado como exemplo é um ambiente onde se tem o SGBD e o servidor de aplicação concentrados em um servidor dedicado da empresa e as aplicações clientes distribuídas pelo globo, acessando-o via Web.

Nota do DevMan

REST, acrônimo de Representational State Transfer (Transferência de Estado Representacional), é uma forma de projetar aplicativos Web fracamente acoplados, cujos recursos são nomeados e acessíveis diretamente através de uma URI (Identificador Uniforme de Recursos) exclusiva para cada recurso.

JSON, acrônimo de Java Script Object Notation, é uma alternativa simplista ao XML e caracteriza-se por ser um formato para a troca de dados entre aplicações heterogêneas. Tem suporte a uma enorme gama de linguagens, tais como: Delphi, C#, Java, PHP, Python, Ruby, C++ entre outras.

AJAX é o acrônimo de Asynchronous Javascript And XML, é uma metodologia computacional para tornar páginas Web mais interativas com o usuário e, para isso, são utilizadas tecnologias como JavaScript e XML, que garantem a troca assíncrona de informações entre as partes Cliente e Servidor.

Nota do DevMan

Load Balance ou, em português, balanceamento de carga, é uma técnica computacional em que se distribui de forma uniforme uma carga de trabalho, entre dois ou mais computadores.

Failover é a capacidade de se passar automaticamente o processamento de um computador servidor em falha para outro computador servidor. Esta técnica computacional é largamente utilizada em sistemas ou redes que necessitam de disponibilidade contínua de recursos, onde se espera um alto grau de confiabilidade.

Independência

Uma vez construído o servidor DataSnap, este poderá ser acessado por qualquer aplicação Cliente, independente de linguagem de programação, desde que tenha suporte a REST, JSON ou JavaScript, que são as tecnologias necessárias para se acessar os métodos do servidor de aplicação. Outra independência obtida é com a parte de banco de dados. No modelo multicamadas, o banco serve apenas como mero repositório de dados, ou seja, independe de recursos específicos de determinado SGBD.

A aplicação de exemplo

A intenção da parte prática deste artigo é mostrar a capacidade em se trabalhar com vários clientes distintos, a partir de um servidor de aplicação DataSnap centralizado. Para tal, será desenvolvida uma aplicação real, multicamadas, de uma empresa fictícia, onde será abordado o processo de autenticação de um usuário no sistema. Explicando melhor, tal empresa possui em uma máquina servidora dedicada um servidor de aplicação DataSnap, do tipo VCL Forms que, entre seus métodos de negócio, está o de autenticação, que verifica numa base de dados Firebird se determinado usuário possui permissão para acesso ou não ao sistema. Os usuários poderão ser funcionários da própria empresa ou fornecedores. Os funcionários utilizarão diariamente o sistema na rede local. Os fornecedores irão utilizar algumas telas de consultas na aplicação, acessando o sistema de qualquer ponto do globo pela Internet, através de 3 opções: uma aplicação VCL for Web, uma página PHP ou uma página ASP.NET com AJAX. O cadastro destes usuários será efetuado pelo setor de Recursos Humanos da empresa, através de uma aplicação cliente VCL Forms local. O esboço desta proposta pode ser visto na **Figura 2**, que servirá como um guia no desenrolar do artigo.

Banco de dados

Como mencionado, o sistema contará com um banco de dados Firebird, onde serão armazenados os usuários cadastrados na empresa. Foi adotada a mais recente versão deste SGBD, a 2.5. Em termos de estrutura, será extremamente simplista, sendo necessária apenas uma tabela. Portanto, com o Firebird instalado e utilizando algum software de gerenciamento do banco de sua preferência, execute o script da **Listagem 1**.

Este script cria efetivamente o banco e a estrutura da tabela necessária para o exemplo. Vale ressaltar que onde aparece o texto [caminho_do_banco] você deverá substituí-lo por um caminho de sua preferência em seu disco.

Servidor de Aplicação

Consideramos um novo projeto no Delphi XE do tipo DataSnap Server. Nesta

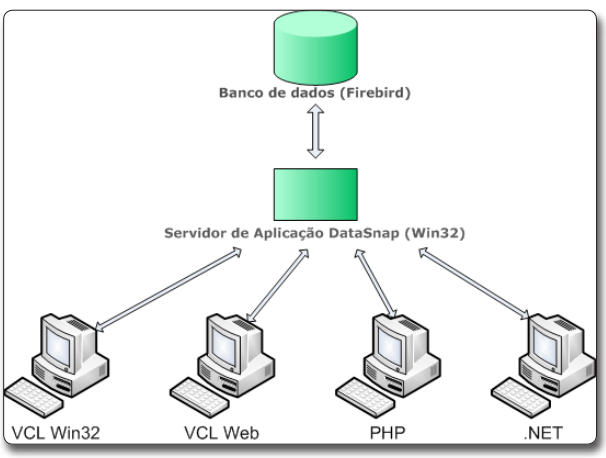


Figura 2. Estrutura da solução proposta

Listagem 1. Script do banco de dados

```
SET SQL DIALECT 3;
SET NAMES WIN1252;
SET CLIENTLIB 'fbclient.dll';
CREATE DATABASE '[caminho_do_banco]\empresa.fdb'
USER 'SYSDBA' PASSWORD 'masterkey'
PAGE_SIZE 8192
DEFAULT CHARACTER SET WIN1252;

CREATE TABLE USUARIOS (
   USUARIO_ID  INTEGER NOT NULL,
   NOME        VARCHAR(80) NOT NULL,
   LOGIN       VARCHAR(10) NOT NULL,
   SENHA       VARCHAR(10) NOT NULL
);

ALTER TABLE USUARIOS ADD CONSTRAINT PK_USUARIOS PRIMARY KEY (USUARIO_ID);
```

etapa, há duas observações a serem feitas. A primeira diz respeito à escolha do tipo de projeto do servidor no Wizard, que deverá ser do tipo VCL Forms Application, conforme a proposta inicial. A segunda é com relação aos protocolos de acesso ao servidor. Como mencionado, alguns clientes estarão na plataforma Web e outros em ambiente Desktop, portanto o servidor tem suporte a conexões tanto via protocolo HTTP, quanto TCP/IP (**Figura 3**).

Com o Delphi 2010 com uma série de novidades foi incluída na área do DataSnap. Foi adicionada à arquitetura DataSnap suporte a transporte de dados sobre HTTP, de forma que a comunicação entre o cliente e o servidor DataSnap além de poder ser feita via TCP/IP utilizando DBExpress, pode ser feita também via HTTP sem a necessidade de nenhuma tecnologia de terceiros. A partir desta versão é possível desenvolver servidores de aplicação de forma que estes servidores possam ser consumidos por um cliente desenvolvido por praticamente qualquer linguagem de programação graças à arquitetura REST e aos dados trafegados em notação JSON.

Os clientes acessam o servidor tanto para cadastrar novos usuários quanto para fazer sua autenticação no sistema. Nestes dois casos mostra-se a necessidade do servidor de se comunicar com a camada de dados, ou seja, com o banco propriamente dito. A conexão com o Firebird se chama EMPRESA, feita no Data Explorer do IDE. Dessa forma, haverá uma conexão pré-configurada para o banco de dados, fazendo uso do driver dbExpress exclusivo para o SGBD.

O container ServerMethods1 contém a classe que será exposta ao meio externo, com os métodos e objetos disponíveis para serem acessados pela parte cliente da solução. A **Figura 4** mostra os componentes utilizados.

Por não fazer parte do escopo do artigo (não estamos seguindo o estilo passo a passo), a configuração destes componentes não será abordada em detalhes, apenas algumas considerações serão feitas. No SQLConnection, em sua propriedade ConnectionName, foi utilizada a conexão EMPRESA. O TSQLDataSet está vinculado a SQLConnection através de sua propriedade de mesmo nome, bem como seu CommandText que contém a seguinte instrução SQL:

```
select * from USUARIOS
```

Método de negócio

Em ServerMethods1 temos um método de negócio que faz a autenticação de um usuário no sistema (**Listagem 2**). Este método é consumido pelas aplicações clientes apresentadas posteriormente neste artigo.

Nessa listagem, na linha 4 é declarada a variável de apoio do tipo TSQLQuery que será usada para realizar a consulta no banco de dados. Na linha 6 é definida uma constante do tipo string, que representa o comando SQL a ser executado pela consulta, que verifica se determinado usuário está cadastrado no sistema com base no Login e Senha informados. Das linhas 9 a 15, o objeto TSQLQuery é instanciado e recebe os valores de suas propriedades SQLConnection, SQL e Params. Esses parâmetros se referem às informações de Login e Senha, cujos valores são obtidos pelos valores dos parâmetros de entrada do próprio método. Na linha 17 há o retorno da função que resulta em True quando o usuário informado for encontrado na base de dados do sistema e False caso contrário.

O servidor de aplicação do exemplo possui todos os recursos necessários para ser utilizado, contudo, para deixá-lo totalmente funcional, é necessário o código da **Listagem 3**, que apenas controla a conexão do componente TSQLConnection com o banco. Apenas é garantida programaticamente a ativação da conexão do servidor de aplicação com o banco de dados na criação do módulo ServerMethods, bem como a desativação da mesma quando o módulo é destruído.

Conexão DataSnap no Data Explorer

No Data Explorer a conexão EMPRESA-CLIENT-TCP realiza a conexão com o

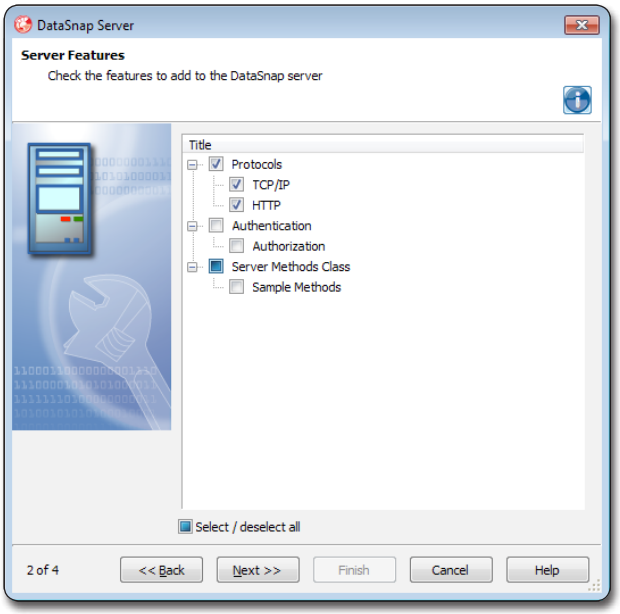


Figura 3. Suporte a TCP/IP e HTTP no Servidor de Aplicação

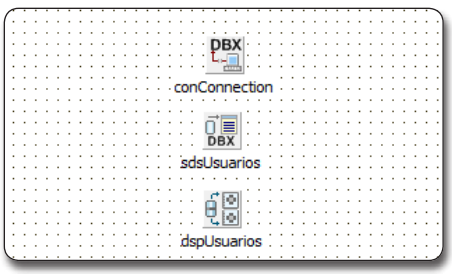


Figura 4. Componentes de acesso a dados usados no servidor de aplicação

Listagem 2. Método de autenticação de usuários

```
01 function TServerMethods1.Autentica
     Usuario(ALogin, ASenha: string):
02 Boolean;
03 var
04   Qry: TSQLQuery;
05 const
06   SQL: string = 'select USUARIO_ID from
       USUARIOS where LOGIN = :login ' +
07             'and SENHA = :senha';
08 begin
09   Qry := TSQLQuery.Create(nil);
10   try
11     Qry.SQLConnection := conConnection;
12     Qry.SQL.Text := SQL;
13     Qry.Params[0].AsString := ALogin;
14     Qry.Params[1].AsString := ASenha;
15     Qry.Open;
16
17     Result := not Qry.IsEmpty;
18   finally
19     FreeAndNil(Qry);
20   end;
21 end;
```

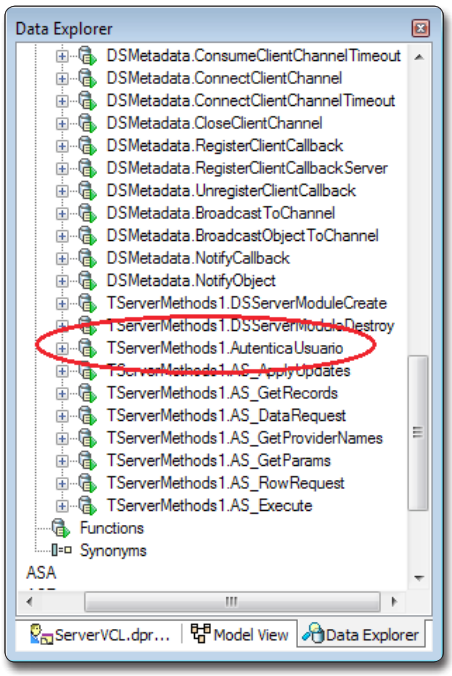


Figura 5. Método AutenticaUsuario devidamente mapeado no Data Explorer

Listagem 3. Eventos OnCreate e OnDestroy da unit ServerMethods

```
procedure TServerMethods1.DSServerModuleCreate(Sender: TObject);
begin
 conConnection.Connected := True;
end;

procedure TServerMethods1.DSServerModuleDestroy(Sender: TObject);
begin
 conConnection.Connected := False;
end;
```

servidor de aplicação, ela é configurada para suportar o protocolo TCP/IP. Em Procedures estão todos os métodos do servidor disponíveis para os clientes, inclusive o método de autenticação criado anteriormente, conforme pode ser visto na **Figura 5**.

Existe outra conexão, configurada nos mesmos moldes, porém nomeada como EMPRESA-CLIENT-HTTP, com suporte ao protocolo HTTP, direcionada para a porta e host do servidor que, por padrão, tem os valores 8080 e localhost, respectivamente. Lembrando que qualquer alteração nas configurações do servidor deverá ser refletida na configuração desta conexão. Este tipo de conexão será usado por clientes que acessam o servidor via Web.

Aplicação Cliente – VCL Forms

A primeira aplicação cliente é do tipo VCL Forms (**Figura 6**), que realiza a manutenção do quadro de usuários da empresa. Vale lembrar que aqui o processo de manutenção é cabível como os passos para editar, excluir e principalmente adicionar novos usuários ao sistema. Este tipo de aplicação fica disponível internamente na empresa, em sua rede local.

Na **Listagem 4** temos a codificação para cdsUsuarios, em seus eventos AfterDelete e AfterPost. O código é bastante simples e tem a função de aplicar as alterações efetivamente na base de dados após a chamada dos métodos Delete e Post do TClientDataSet.

A codificação dos botões, na mesma listagem, refere-se às ações de manipulação dos dados da tabela do banco de dados. Os botões Inserir, Alterar e Excluir chamam respectivamente os métodos Append, Edit e Delete do TClientDataSet que são relativos às ações de cada um. Já para o botão Salvar, é chamado o método Post e a seguir é exibida uma mensagem de sucesso em tela, através do ShowMessage.

Para finalizar, os eventos OnCreate e OnDestroy do formulário servem para manter o bom funcionamento da aplicação, ativando e desativando a conexão com o servidor conforme o form é criado ou destruído.

Feito isso, compilando e executando a aplicação, basta fazer o cadastro de alguns usuários para os futuros testes de autenticação a serem realizados pelas demais aplicações clientes. Para visualizar os lançamentos dos dados basta acessar a tabela pelo próprio Data Explorer. No exemplo foi cadastrado um usuário com as credenciais Login = admin e Senha = 123, que será usado para testar as aplicações clientes daqui em diante.

Listagem 4. Eventos AfterDelete e AfterPost do ClientDataSet

```
procedure TfrmCadUsuarios.cdsUsuariosAfterDelete(DataSet: TDataSet);
begin
 cdsUsuarios.ApplyUpdates(0);
end;

procedure TfrmCadUsuarios.cdsUsuariosAfterPost(DataSet: TDataSet);
begin
 cdsUsuarios.ApplyUpdates(0);
end;

procedure TfrmCadUsuarios.btnAlterarClick(Sender: TObject);
begin
 cdsUsuarios.Edit;
end;

procedure TfrmCadUsuarios.btnExcluirClick(Sender: TObject);
begin
 cdsUsuarios.Delete;
end;

procedure TfrmCadUsuarios.btnInserirClick(Sender: TObject);
begin
 cdsUsuarios.Open;
 cdsUsuarios.Append;
end;

procedure TfrmCadUsuarios.btnSalvarClick(Sender: TObject);
begin
 cdsUsuarios.Post;
 ShowMessage('Registro salvo com sucesso!');
end;

procedure TfrmCadUsuarios.FormCreate(Sender: TObject);
begin
 conConnection.Connected := True;
end;

procedure TfrmCadUsuarios.FormDestroy(Sender: TObject);
begin
 conConnection.Connected := False;
end;
```

Aplicação Cliente – VCL for the Web

A partir de agora, apresentamos a aplicação cliente para a plataforma Web, que fica disponível para os usuários que estão no ambiente externo à empresa, no caso os fornecedores. O projeto usado é do tipo VCL for Web, utilizando o template VCL for Web Application Wizard. O exemplo foi construído utilizando o tipo Stand Alone Application.

Dentre os arquivos criados, será aberto o container UserSession, que nada mais é que uma espécie de Data Module para este tipo de projeto. A conexão pode ser configurada arrastando-se para dentro do UserSession (a partir do Data Explorer) a conexão DataSnap EMPRESA-CLIENT-HTTP, dessa forma já haverá uma conexão válida com o servidor através de um TSQLConnection. O TSQLServerMethod da paleta dbExpress, ligado ao TSQLConnection, aponta para o método remoto AutenticaUsuario criado anteriormente. A **Figura 7** mostra o form desta aplicação. A **Listagem 5** mostra o evento OnClick do botão Autenticar, onde é feita a chamada ao método remoto de autenticação.

Nesta listagem, entre as linhas 4 e 8, são passados os parâmetros, provindos dos controles visuais edtLogin e edtSenha, para o componente SqlServerMethod1, que mapeia o método do servidor. A método é executado no servidor de aplicação através do ExecuteMethod. Na linha 10 é testado o seu retorno e, de acordo com o resultado, uma mensagem de sucesso ou falha na autenticação é exibida ao usuário através do label lblResult, nas linhas 11 e 13.

Realizando alguns testes com o usuário de exemplo cadastrado anteriormente é possível comprovar a funcionalidade do método definido no servidor e consumido por esta aplicação cliente.

Aplicação Cliente – .NET (Delphi Prism XE)

A aplicação do tipo .NET usa o Delphi Prism XE para o seu desenvolvimento. O projeto utilizado é do tipo ASP.NET Web Application. Uma importante configuração realizada é a conexão DataSnap no Server Explorer do IDE (**Figura 8**), que é similar ao Data Explorer do Delphi. A conexão é configurada conforme a autenticação

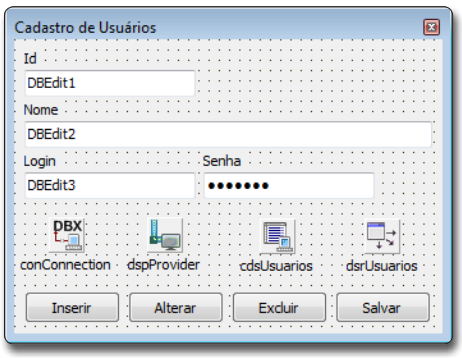


Figura 6. Formulário da aplicação cliente VCL Forms

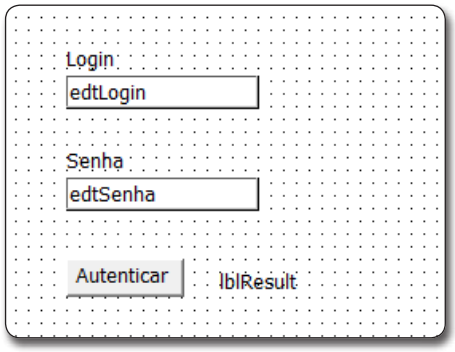


Figura 7. Visual da aplicação VCL for the Web

HTTP definida anteriormente no Data Explorer do Delphi. As classes Proxy Client provêem o acesso aos métodos do servidor de aplicação (**Figura 9**).

Um proxy, em sua forma mais geral, é uma classe que funciona como uma interface para outra classe. Em padrão de projeto a função de uma classe proxy é prover um representante para outro objeto de modo a controlar o acesso a este. No caso do DataSnap podemos dizer que temos um proxy remoto, que é quando a classe proxy provê um representante local para um objeto em um espaço de endereçamento diferente, neste caso o servidor de aplicação.

A **Figura 10** apresenta o layout proposto para a página Default.aspx, que é muito semelhante ao que foi feito na aplicação cliente anterior. Vale notar que os controles foram inseridos sobre um UpdatePanel que, junto do controle ScriptManager, são os responsáveis pela parte AJAX da aplicação. A seguir na **Listagem 6** é ilustrado o evento OnClick do botão Autenticar.

No código na linha 4 é declarada uma variável do tipo da classe Proxy gerada, onde será armazenada uma instância da mesma. Na linha 7 há a variável que representará a conexão DataSnap criada no Server Explorer. Das linhas 9 a 11, é criado um objeto que recebe uma instância da conexão a partir de uma string de conexão. Em seguida, a conexão é ativada

Listagem 5. Evento OnClick do botão Autenticar - Delphi

```
01 procedure TIWForm3.btnAutenticarClick(Sender: TObject);
02 begin
03   //adicionar a unit ServerController ao uses
04   with UserSession.SqlServerMethod1 do
05   begin
06     Params[0].AsString := edtLogin.Text;
07     Params[1].AsString := edtSenha.Text;
08     ExecuteMethod;
09
10     if Params[2].AsBoolean = True then
11       lblResult.Caption := 'Usuário autenticado'
12     else
13       lblResult.Caption := 'Usuário não autenticado';
14   end;
15 end;
```

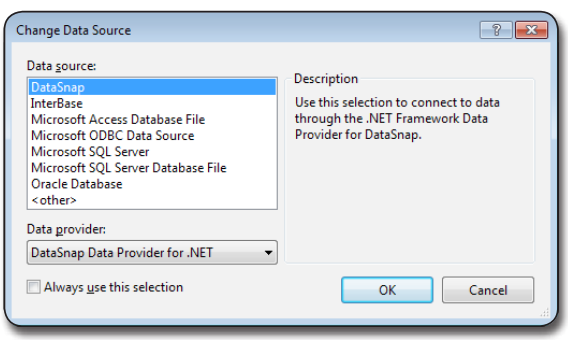


Figura 8. DataSource do tipo DataSnap

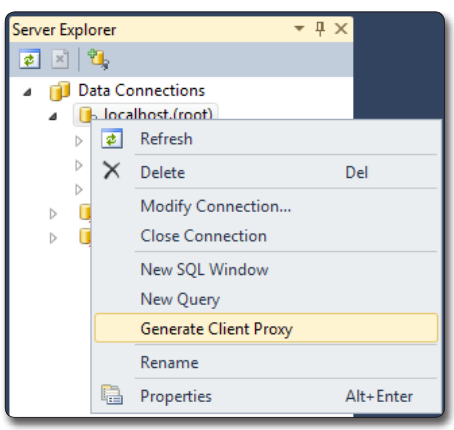


Figura 9. Classe Proxy cliente no Delphi Prism

pelo método Open. Na linha 13, a variável ClientPrx recebe a instância da classe Proxy que é criada passando-se a conexão ativa no momento. Na linha 14 é invocado o método remoto AutenticaUsuario através da instância da classe Proxy cliente, passando para o método os valores digitados em tela em txtLogin e txtSenha. Nas linhas 15 e 17 é mostrado ao usuário da aplicação o resultado de sucesso ou falha na autenticação de acordo com o resultado obtido no método remoto. Na linha 19 a conexão com o servidor é encerrada.

Aplicação Cliente – PHP (RadPHP XE)

A aplicação cliente PHP utiliza a ferramenta RadPHP XE, anteriormente conhecida como Delphi for PHP. A aplicação usa um template chamado RCPL Application – PHP, com um item DataSnap REST Client Module (**Figura 11**). Uma observação: este objeto é responsável por gerar a classe Proxy cliente para que a aplicação PHP possa consumir os métodos do servidor.

A tela de configuração da **Figura 12** contém os dados pertinentes à conexão com o servidor DataSnap. Aqui há uma observação importante a se fazer. Em seu parâmetro

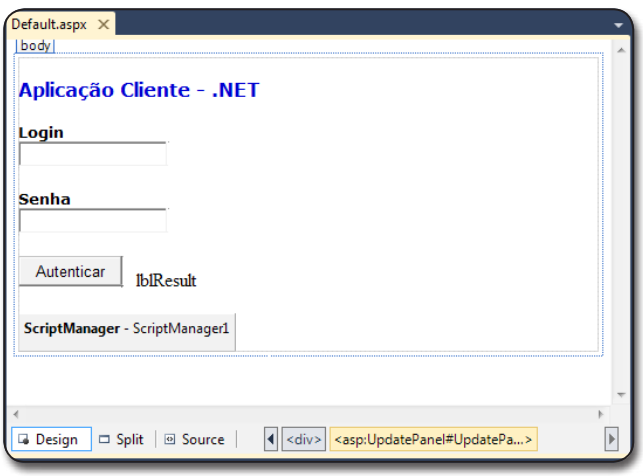


Figura 10. Layout da aplicação cliente Delphi Prism

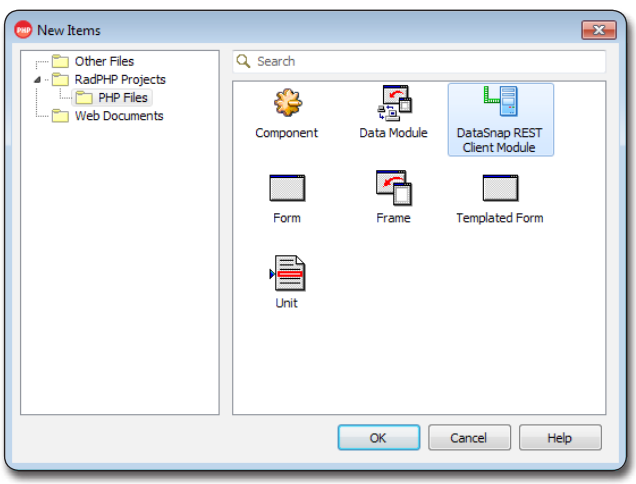


Figura 11. DataSnap REST Client Module no RadPHP XE

denominado Language, é onde é definido se a classe Proxy cliente será gerada na linguagem PHP ou JavaScript. Será usada a primeira opção. A **Figura 13** ilustra o modelo de layout da unit1.php. Para o evento OnClick do botão Autenticar, segue sua codificação na **Listagem 7**.

No código, inicialmente na linha 3 é declarada uma variável global referente ao ClientModule criado junto da classe Proxy cliente. A seguir, na linha 6, o método do servidor é invocado, passando como parâmetros os valores digitados em tela nos controles TEdit. Seu retorno é armazenado na variável "autenticado". O tipo de retorno deste método da classe Proxy no RadPHP é um objeto do tipo DSObject. Portanto, na linha 10, é verificado se o retorno condiz com o tipo de objeto esperado. Na linha 11, é testada a propriedade result deste objeto, que contém o valor booleano de retorno do método remoto. Caso o usuário tenha sido autenticado ou não, uma mensagem é exibida no label lblResult nas linhas 12 e 13. Feito isso, mais uma vez executando a aplicação é possível comprovar sua

Nota do DevMan

O AJAX foi uma das grandes novidades e carro chefe da chamada Web 2.0. No ASP.NET ele surgiu como dois simples controles: o ScriptManager e o UpdatePanel. Aplicar uma interface AJAX em uma aplicação ASP.NET Web Forms, na maioria das vezes significava apenas incluir um ScriptManager no topo da página e colocar os controles que realizam Postback dentro de UpdatePanels. Tratando algumas possíveis exceções, com apenas isso já podemos dizer que nossa página está com AJAX. A seguir, veja uma descrição resumida dos principais recursos desses controles:

ScriptManager: Corresponde a uma das principais partes do ASP.NET AJAX. Ele é responsável por gerenciar os recursos de AJAX numa página ASP.NET, incluindo baixar os JavaScript da biblioteca AJAX para o browser e coordenar atualizações parciais nas páginas que são habilitadas com uso do UpdatePanel. É muito importante garantir que um ScriptManager esteja colocado na página ASPX antes de qualquer outro controle que necessite dele, como ocorre no caso de um UpdatePanel.

UpdatePanel: São utilizados com o ScriptManager para permitir a atualização parcial de uma página. A possibilidade de atualizar parcialmente o conteúdo de uma página Web melhora a experiência do usuário por uma série de motivos: 1) Diminui o volume de informações trocadas entre o servidor Web e o browser; 2) Reduz o efeito de "piscar" em relação a uma atualização completa do conteúdo; 3) A interface não é posicionada no início da página em cada retorno ao servidor Web; 4) O UpdateProgress pode ser utilizado em conjunto com o UpdatePanel de modo a apresentar uma informação para o usuário durante a comunicação com o servidor Web.

Listagem 6. Evento Click do botão Autenticar - Prism

```
01 method _Default.btnAutenticar_Click(sender: System.Object;
02 e: System.EventArgs);
03 var
04  ClientPrx: ClientProxy.TServerMethods1Client;
05
06  //Adicionar o namespace Borland.Data.*
07  Conexao: TAdoDbxDatasnapConnection;
08 begin
09   Conexao := new TAdoDbxDatasnapConnection('
10   communicationprotocol=http;port=8080;hostname=localhost');
11   Conexao.Open;
12   try
13    ClientPrx := ClientProxy.TServerMethods1Client.Create(Conexao);
14    if ClientPrx.AutenticaUsuario(txtLogin.Text, txtSenha.Text) then
15     lblResult.Text := 'Usuário autenticado'
16    else
17     lblResult.Text := 'Usuário não autenticado';
18   finally
19    Conexao.Close;
20  end;
21 end;
```

funcionalidade em consumir um método remoto do servidor.

Conclusão

O assunto multicamadas, é uma constante nos temas dos artigos desta revista, dada sua importância no cenário do desenvolvimento. Neste artigo foram mostradas e reforçadas novamente algumas das vantagens em se trabalhar com aplicações desse tipo, tais como centralização das regras de negócio, escalabilidade, clara divisão de responsabilidades, entre outras. Tudo isso é conseguido de forma muito facilitada no Delphi, utilizando seu Framework DataSnap, que gradativamente a cada lançamento de uma nova versão da IDE sofre melhorias significativas. A prova mais recente disso é que com o Delphi XE, o DataSnap apresentou entre suas novidades, um maior suporte nativo para a construção de clientes em outras tecnologias: PHP, JavaScript, e C++. Isso aumentou ainda mais o leque de possibilidades das aplicações, sejam elas comerciais ou não, uma vez que se obtém uma independência de plataforma (Desktop, Web e Mobile) e linguagem.

Com base no que foi mostrado aqui neste artigo, alguns desafios podem e deverão ser alcançados. Cito dois deles: desenvolvedores que ainda possuem suas aplicações no modelo mais tradicional de duas camadas, podem se encorajar definitivamente a migrar para multicamadas DataSnap e aqueles que já possuem suas aplicações neste modelo muito provavelmente, de agora em diante, já irão trabalhar com a ideia de aumentar sua gama de aplicações clientes.

Alongando um pouco mais, ainda é possível aproveitar a estrutura construída neste artigo e desenvolver outros clientes, utilizando, por exemplo, uma classe Proxy cliente em JavaScript, que pode ser facilmente criada no RadPHP XE, ou até mesmo utilizando C++, através do C++ Builder XE.

Listagem 7. Evento Click do botão Autenticar - PHP

```
01 function btnAutenticarClick($sender, $params)
02  {
03   // adicionar ClientModuleUnit1 ao uses
04   global $ClientModuleDataModule1;
05
06   $autenticado =
07 $ClientModuleDataModule1->ServerMethods1Client-06>AutenticaUsuario(
08 $this->edtLogin->Text, $this->edtSenha->Text);
09
10   if ($autenticado instanceof DSObject)
11    if ($autenticado->result)
12     $this->lblResult->Caption = 'Usuario autenticado';
13    else
14     $this->lblResult->Caption = 'Usuario nao autenticado';
15 }
```

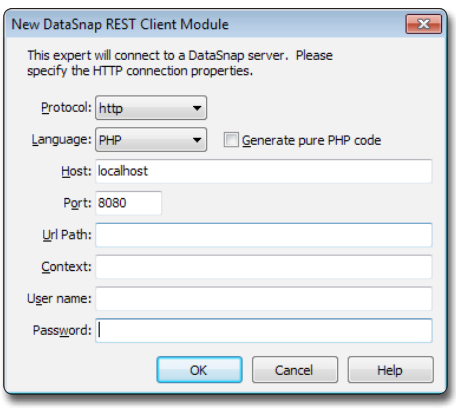


Figura 12. Configurando a conexão DataSnap no RadPHP XE

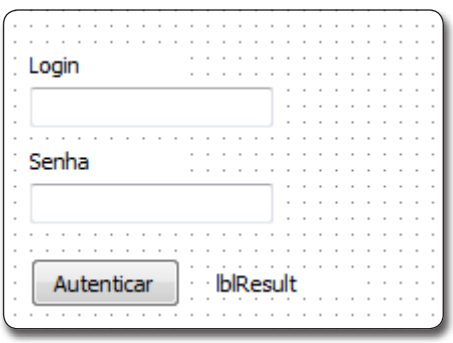


Figura 13. Layout da aplicação cliente PHP

Fabrício Hissao Kawata
fabricio.kawata@bol.com.br
Formado em Processamento de Dados pela FATEC-TQ. Atua como Analista Programador Delphi há 5 anos. Atualmente escreve artigos sobre Delphi para revistas especializadas além de participar do programa Beta Tester Delphi Prism, da Embarcadero.

Embarcadero RAD Studio XE Architect - Trial
https://downloads.embarcadero.com/free/rad_studio

Delphi Prism XE - Trial
https://downloads.embarcadero.com/free/delphi_prism

RadPHP XE – Trial
https://downloads.embarcadero.com/free/radphp

Dê seu feedback sobre esta edição!

A Clubedelphi tem que ser feita ao seu gosto. Para isso, precisamos saber o que você, leitor, acha da revista!

Dê seu voto sobre este artigo, através do link:

www.devmedia.com.br/clubedelphi/feedback

XML Data Binding

Manipule arquivos XML de forma

No cenário tecnológico atual existe uma imensa quantidade de soluções desenvolvidas em inúmeras linguagens de programação e uma demanda cada vez maior de integração entre esses sistemas. Com base nessa necessidade, o padrão de arquivo XML (eXtensible Markup Language) passou a ser utilizado por inúmeros projetos, como por exemplo a NFe (Nota Fiscal Eletrônica) e a grande maioria de serviços disponibilizados via Web Service.

A XML veio para substituir os "antigos" arquivos em formato texto que normalmente eram utilizados para troca de informações entre sistemas. Com a utilização de arquivos texto, havia muitos problemas todas as vezes que algum tipo de mudança ocorria nos campos que compunham o arquivo. Quem nunca recebeu uma solicitação para aumentar o tamanho do campo X ou Y e esses campos faziam parte de alguma rotina de exportação baseada em texto. Quando isso ocorria, toda a rotina de exportação e importação tinha que ser revista, gerando grandes transtornos.

Com a utilização de arquivo XML esse problema não ocorre, pois a XML não delimita o tamanho do conteúdo dos campos. Isso facilita muito a vida do desenvolvedor que não precisa se preocupar com as rotinas de exportação e importação já escritas.

Além da utilização em integração de sistemas a XML também é amplamente utilizado em arquivos de configuração e em inúmeras outras tecnologias, tais com Sistemas de Informação Geográficas (SIG), Backup, dispositivos móveis etc.

Resumo DevMan

De que trata o artigo:

Geração de interfaces que encapsulam a complexidade do acesso aos dados dos arquivos XML. Fazendo uso destas interfaces o desenvolvedor ganha uma forma simples, transparente, elegante e eficiente de manipular o conteúdo desses arquivos que são utilizados em larga escala no atual cenário de desenvolvimento.

Para que serve:

Para facilitar o trabalho do desenvolvedor no momento da leitura ou geração de arquivos no formato XML. Essa ferramenta gera a partir de um modelo, uma série de interfaces responsáveis pelo acesso aos dados do XML. A utilização dessa ferramenta é uma forma de abstrair a complexidade e permitir ao desenvolvedor acesso aos dados da mesma forma que ele acessa propriedades no Delphi.

Em que situação o tema é útil:

Na integração entre sistemas que fazem uso de arquivos XML para troca de informação, podemos citar como exemplos a NFe (Nota Fiscal Eletrônica). Também podemos utilizar as interfaces geradas pela ferramenta para consumir dados vindos de um Web Service, por exemplo. De maneira geral podem ser geradas interfaces para acesso a qualquer arquivo XML, não importando a sua complexidade, com base nisso a sua utilização é imensamente ampla, podendo facilitar a vida do desenvolvedor em incontáveis situações.

XML Data Binding:

No decorrer do artigo vamos falar um pouco sobre o formato XML, entender o funcionamento do XML Data Binding, como ele abstrai toda a complexidade de manipulação de arquivos XML, gerar as interfaces necessárias para interagir com um arquivo XML e por fim, vamos utilizá-las em um pequeno exemplo, para demonstrar como é simples, elegante e eficiente o acesso a arquivos no formato XML.

O formato XML

O formato XML surgiu em meados de 1990, é uma linguagem de marcação baseada em texto que combina a flexibilidade do formato SGML com a simplicidade do formato HTML. O XML é um formato que permite a criação de dados de maneira organizada e hierárquica.

Sendo o XML um formato que não depende de plataforma de hardware ou software, esse arquivo pode facilmente ser gerado por uma aplicação rodando em Linux e ser lido ou importado por outra aplicação rodando em Windows ou outro sistema qualquer sem nenhuma modificação. Por esse motivo ele se tornou amplamente utilizado. Algumas características do formato XML:

- Separação do conteúdo da formatação;
- É de fato um padrão;
- Baseado em texto;
- É simples para ser lido por um ser humano;
- Possibilidade de criação de tags sem limitação;
- Criação de arquivos para validação de estrutura (chamados DTDs ou XmlSchema);
- Independência no tamanho dos campos.

A **Listagem 1** demonstra um exemplo de arquivo no formato XML que será utilizado como exemplo no decorrer do artigo.

Composição de um arquivo XML

Um documento XML é composto pelos seguintes tipos de marcações, sendo que

fácil e rápida

JÉFFERSON RICARDO ZUCHI

a maioria deles também é suportada em arquivos HTML e XHTML:

- Elementos;
- Atributos;
- Comentários, no mesmo formato de HTML e XHTML;
- Entidades;
- Instruções para processamento (Processing Instructions);
- Seções CDATA;
- Comentários.

As entidades no formato XML são utilizadas para representar caracteres especiais de marcação (<,&,>,"). As instruções de processamento, também conhecidas como PI, são definidas com o formato <?nome dados?>. A primeira marcação de um arquivo XML é justamente um PI onde se identifica a versão do XML e o conjunto de caracteres do mesmo.

As seções CDATA delimitam o conteúdo que deve ser ignorado pelo analisador sintático do documento. A sintaxe para definição da tag CDATA é:

<![CDATA[texto qualquer terminado com]]>

Para adicionar comentários em um arquivo XML deve-se utilizar a sintaxe:

<!-- seu comentário pode ser escrito aqui -->

XML Data Binding – Conceito, Funcionamento e Assistente

O conceito de binding de arquivos XML refere-se ao mapeamento das informações contidas no arquivo XML em objetos armazenados na memória do computador, desta forma, as aplicações fazem acesso aos objetos em memória e não diretamente ao arquivo XML. O conceito de XML Data Binding é muito semelhante ao conceito de mapeamento objeto-relacional.

Antes de colocar a mão na massa e gerar as interfaces para manipulação de arquivos XML é interessante entender como a "mágica" do XML Data Binding acontece. Todo o segredo está contido na unit XMLIntf.pas, disponibilizada com o Delphi.

Nota do DevMan

O formato SGML surgiu na década de 60 e foi desenvolvido pelos pesquisadores da IBM C. Goldfarb, E. Mosher e R. Lorie, visando facilitar o intercâmbio e a manipulação de documentos, independente de sistema operacional e formato de arquivos. A opção foi utilizar um sistema de "Marcação Generalizada" (Generalized Markup). Os objetivos básicos do formato eram:

- a marcação deveria descrever a estrutura do documento e outros atributos do mesmo em vez de especificar o processamento a ser feito no mesmo;

- a marcação deve ser definida rigorosamente, de forma que sistemas formais como programas possam ser usados para processar o documento.

O SGML surgiu aproximadamente na mesma época que a internet e o sistema UNIX, nessa época não se fazia uso generalizado destas duas tecnologias.

Listagem 1. Exemplo de arquivo XML

```
<?xml version="1.0" encoding="UTF-8"?>
<curriculo>
 <InformacaoPessoal>
  <DataNascimento>09/07/1979</DataNascimento>
  <Nome>Jefferson Ricardo Zuchi</Nome>
  <Contato>
   <Endereco>
    <Rua>Rua Rio Grande do Sul</Rua>
    <Numero>2403</Numero>
    <Bairro>Boa Vista</Bairro>
    <Cidade>Barueri</Cidade>
    <CEP>99999-999</CEP>
    <UF>SP</UF>
   </Endereco>
   <Telefones>
    <residencial>(11) 1111-1111</residencial>
    <comercial>(11) 1212-1212</comercial>
    <celular>(11) 1313-1313</celular>
   </Telefones>
   <email>jefferson@sygnux.com.br</email>
  </Contato>
  <Sexo>Masculino</Sexo>
 </InformacaoPessoal>
 <objetivo>Atuar na area de Desenvolvimento</objetivo>
 <Experiencias>
  <Experiencia>
   <Cargo tempo="Atual">Analista de Sistemas Pleno</Cargo>
   <Empregador NomeFantasia="AMIL"
        CNPJ="11.111.111/1111-11">
   </Empregador>
  </Experiencia>
  <Experiencia>
   <Cargo tempo="10 Anos">Analista de Sistemas Pleno</Cargo>
   <Empregador NomeFantasia="Sygnux Software"
        CNPJ="00.000.000/0000-00">
   </Empregador>
  </Experiencia>
  <Experiencia>
   <Cargo tempo="2 ANOS">Professor</Cargo>
   <Empregador NomeFantasia="FATEC"
        CNPJ="22.222.222/2222-22">
   </Empregador>
  </Experiencia>
 </Experiencias>
 <Formacao>Superior Completo</Formacao>
</curriculo>
```

Esta unit define os métodos que deverão ser implementados para permitir a manipulação dos dados em qualquer arquivo no formato XML. Devido a sua padronização é possível determinar de maneira genérica tudo que será necessário para manipular esse formato de arquivo. Nela são definidas as seguintes interfaces:

- IXMLNode;
- IXMLNodeList;
- IXMLNodeCollection; IXMLDocument.

Cada uma dessas interfaces é implementada por uma classe que possui o código responsável pelo seu funcionamento. A interface não possui codificação em seus métodos (procedures ou functions).

Desta forma o trabalho do XML Data Binding se resume a coletar as informações contidas no arquivo de modelo, que normalmente tem as seguintes extensões (*.dtd, *.xsd, *.xdbou *.xml) e fazer uso de uma certa "inteligência" para implementar os métodos descritos nas interfaces. O XML Data Binding é suficientemente inteligente para detectar padrões como listas e implementar a interface correta para manipulá-las.

Nota do DevMan

Interface é um recurso da Orientação a Objetos, que permite uma grande flexibilidade ao desenvolvedor. De modo simples uma Interface nada mais é que um conjunto de declarações de métodos (nome, parâmetros e tipo de retorno) sem uma implementação. Sendo assim é de responsabilidade do programador que deseja implementar a interface em questão providenciar uma implementação (funcionamento) para os métodos na classe que ele está desenvolvendo. Em outras palavras, as Interfaces definem comportamento e as Classes implementam esse comportamento.

Nota

Sempre dê preferência para gerar as interfaces através do XML Data Binding para os arquivos *.dtd, *.xsd ou o *.xdb. Isso é importante, porque esses arquivos são arquivos de "definição de tipo de documento". Eles possuem as regras de validação/implementação para o arquivo XML, sendo assim a possibilidade de erros de tags e de tipo de dados é muito menor ou quase nula. Quando se faz a geração das interfaces com base em um arquivo XML de modelo, corre o risco de faltar alguma tag que não foi gerada para o arquivo específico. Isso ocorre porque não há obrigação de que as tags contenham dados, assim, pode-se simplesmente omiti-las no arquivo XML. Os arquivos *.dtd e *.xsd (XML Schema) são padrões definidos pelo W3C, veja sessão links.

Criando as Interfaces para manipular o XML

As interfaces serão implementadas de maneira automática pelo XML Data Binding, economizando muitas linhas de código. Para o exemplo será utilizado o arquivo XML da **Listagem 1.** Para ter acesso ao XML Data Binding é necessário ter um projeto aberto, assim, um novo é criado.

Este é o momento mais importante de todo o processo, através do assistente do XML Data Binding (**Figura 1**). É exatamente neste momento que são definidos os tipos de dados que serão utilizados na implementação das interfaces e o tipo de acesso (read/write ou read only).

O XML Data Binding, com base na análise do documento da **Listagem 1**, sugere a melhor opção para cada tag do arquivo, mas é necessário tomar cuidado, pois isso pode render grande dor de cabeça.

Por exemplo, para a tag Número, a sugestão de tipo é Integer, isso é compreensível, pois o valor da tag no arquivo XML é 2403, mas em alguns casos existem números de residências, como por exemplo: 203A ou 203B. Esse tipo de problema é muito comum quando se utiliza o próprio XML como base para a geração das interfaces, uma vez que o assistente analisará somente o valor contido no arquivo XML.

Por esse motivo, a melhor opção é sempre utilizar os arquivos de "definição de tipo de documento" (*.dtd, *.xsd ou o *.xdb), eles contêm não um valor, mas sim a regra que determina qual é o tipo de valor para cada tag.

Para resolver estes pequenos problemas basta modificar o tipo de dado através da opção Data Type. Caso a necessidade seja apenas ler o arquivo XML, a opção de read only deve ser usada, com isso o código gerado será menor e menos "poluído", não serão gerados métodos para incluir valor na tag, somente o método responsável pela leitura. As seguintes interfaces são geradas:

```
IXMLCurriculoType = interface;
IXMLInformacaoPessoalType = interface;
IXMLContatoType = interface;
IXMLEnderecoType = interface;
IXMLTelefonesType = interface;
IXMLExperienciasType = interface;
IXMLExperienciaType = interface;
IXMLCargoType = interface;
IXMLEmpregadorType = interface;
```

Cada nó do arquivo XML foi transformado em uma classe e disponibilizado através

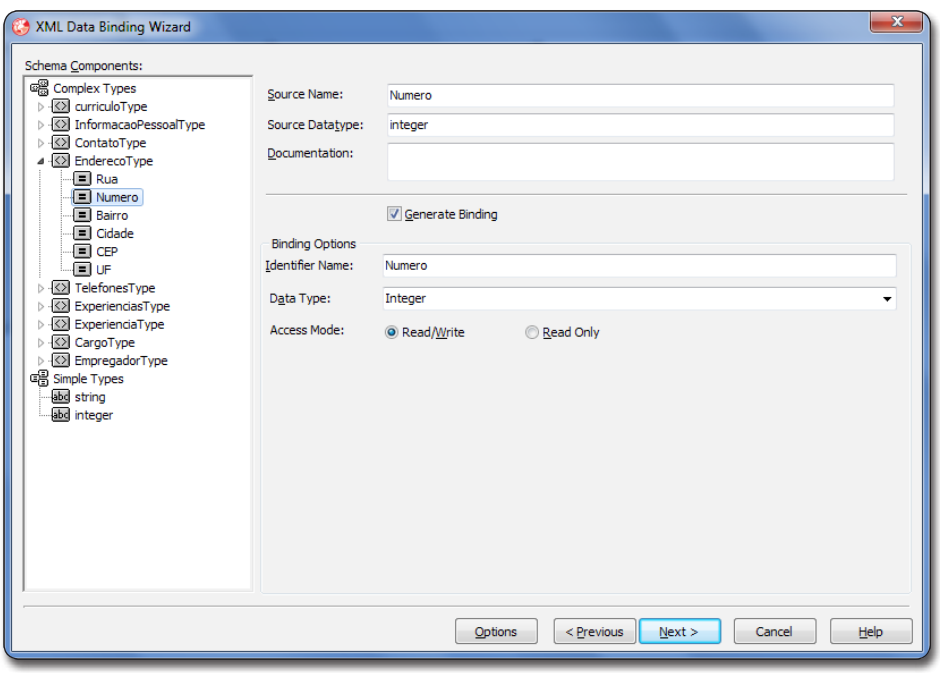


Figura 1. Definição dos tipos de dados no Wizard do XML Data Binding

das interfaces que serão utilizadas para fazer acesso aos dados no nosso XML. A interface topo ou Raiz é a IXMLCurriculoType, detalhada na **Listagem 2**. Ela é o nó (node) mais alto do arquivo XML e é a base para o acesso aos demais nós (nodes). As interfaces de acesso também possuem uma hierarquia que facilita o entendimento e o acesso aos dados.

Já a interface IXMLExperienciasType (**Listagem 3**) faz referência à interface IXMLNodeCollection, responsável por tratar listas. Como foi mencionado o XML Data Binding tem inteligência suficiente para analisar padrões contidos no modelo e definir o tipo de implementação para cada tag do arquivo XML.

A interface IXMLNodeCollection provê métodos para acesso a uma lista, por isso a existência de métodos como Add e Insert. Provavelmente uma pergunta surge: Se isso é uma lista onde está o método Delete, responsável por remover um item? A resposta é simples: O método Delete(Index: Integer) faz referência apenas ao índice da lista, diferente do método Add que neste caso específico retorna um IXMLExperienciaType, sendo assim o método Delete é um método genérico, que está definido na interface IXMLNodeCollection e implementado na classe TXMLNodeCollection. É por esse motivo que não existe a necessidade de definilo na interface IXMLExperienciaType e muito menos implementá-lo diretamente na classe. A reutilização de código é um dos grandes benefícios da Programação Orientada a Objetos.

Além das classes, existem também três métodos globais mostrados no código a seguir, que são responsáveis pelo carregamento e criação do arquivo XML, eles devem ser utilizados para que se possa obter uma instância da classe TXMLCurriculoType, representada pela interface IXMLCurriculoType e a partir daí acessar os conteúdo do XML:

```
function Getcurriculo(Doc: IXMLDocument): IXML-
CurriculoType;
function Loadcurriculo(const FileName: string):
IXMLCurriculoType;
function Newcurriculo: IXMLCurriculoType;
```

Listagem 2. Definição da interface IXMLCurriculoType

```
IXMLCurriculoType = interface(IXMLNode)
  ['{96B0994A-C6D6-4BBE-8B52-2AE4B98F5ED6}']
{ Property Accessors }
function Get_InformacaoPessoal: IXMLInformacaoPessoalType;
function Get_Objetivo: String;
function Get_Experiencias: IXMLExperienciasType;
function Get_Formacao: String;
procedure Set_Objetivo(Value: String);
procedure Set_Formacao(Value: String);
{ Methods & Properties }
property InformacaoPessoal: IXMLInformacaoPessoalType read Get_InformacaoPessoal;
property Objetivo: Stringread Get_Objetivo write Set_Objetivo;
property Experiencias: IXMLExperienciasType read Get_Experiencias;
property Formacao: Stringread Get_Formacao write Set_Formacao;
end;
```

Listagem 3. Implementação da classe IXMLExperienciasType

```
IXMLExperienciasType = interface(IXMLNodeCollection)
  ['{C18A84F3-E9D9-4C88-BFFE-5CE4B5A58C20}']
  { Property Accessors }
function Get_Experiencia(Index: Integer): IXMLExperienciaType;
  { Methods & Properties }
function Add: IXMLExperienciaType;
function Insert(const Index: Integer): IXMLExperienciaType;
property Experiencia[Index: Integer]: IXMLExperienciaType read Get_Experiencia; default;
end;
```

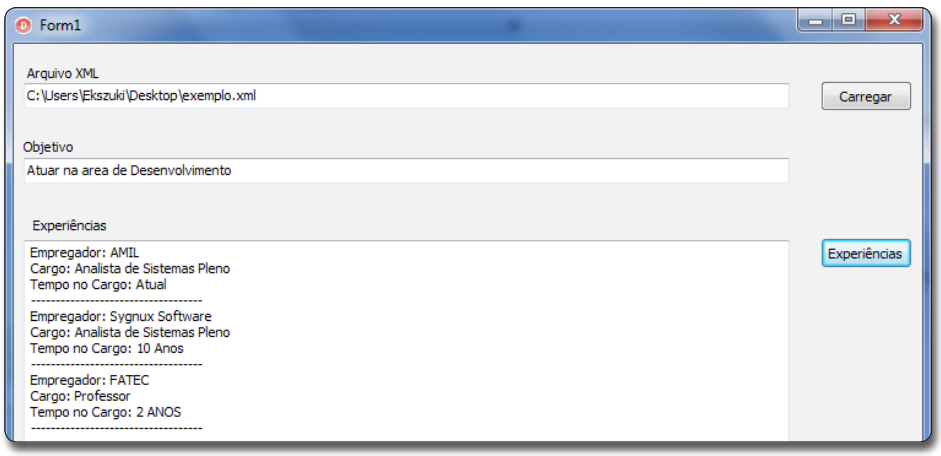


Figura 2. Formulário para apresentação dos dados do XML

Acessando os dados via programação

Agora que já vimos como é o formato XML, e entendemos a "mágica" por trás do XML Data Binding, podemos utilizar as classes que foram geradas e acessar os valores das tags do nosso arquivo XML.

Para utilizar as classes que foram geradas e acessar os valores das tags do arquivo XML, uma interface simples é criada, que irá recuperar os dados do XML e mostrá-los em um formulário, como mostra a **Figura 2**.

Declarando variáveis

Para facilitar o acesso aos dados, algumas variáveis são declaradas, representando os nodes do arquivo XML. Elas são declaradas na seção private, isso é uma boa prática, pois garante que somente os métodos definidos na própria unit terão

NotA

É uma boa prática manter o arquivo gerado pelo XML Data Binding em uma unit separada, para facilitar a manutenção e possíveis atualizações. Talvez isso pareça óbvio, mas já me deparei com situações em que isso não foi feito, agregando dificuldades a uma rotina de atualização que normalmente é muito simples.

acesso a elas. Este é outro conceito da POO, conhecido como encapsulamento e que infelizmente é muito pouco utilizado pelos programadores. As variáveis são:

```
XMLCurriculoType : IXMLCurriculoType;
XMLExperiencias  : IXMLExperienciasType;
XMLExperiencia   : IXMLExperienciaType;
```

Poderiam ser declaradas todas as nossas interfaces, mas isso não é necessário porque elas são acessadas diretamente através da variável XMLCurriculoType. Outro detalhe interessante é que na declaração foi utilizado o tipo da interface (IXMLCurriculoType) e não da classe (TXMLCurriculoType).

Deve-se destacar a importância em utilizar a interface na declaração e não a implementação em si. Utilizar interface é uma forma de encapsular um comportamento. A interface provê uma maneira de esconder qual a classe que realiza uma tarefa específica, definindo o comportamento mínimo que a classe deve suportar. Fazer uso de interface permite ao desenvolvedor modificar a implementação das classes sem modificar as demais classes que as chamam, já que o comportamento em si não mudou, o que muda é a classe que implementa esse comportamento.

Nota do DevMan

De modo geral o encapsulamento garante que somente a própria classe conheça suas variáveis de instância, caso seja necessário o acesso a elas deverá ser feito através da utilização de métodos públicos, ou no caso específico do Delphi, também através das properties.

Essa técnica garante que as regras para manipulação dos valores das variáveis, fiquem centralizadas em uma única classe ou método, evitando o citado "efeito gelatina" onde uma pequena mudança de código afeta todo o resto do sistema, da mesma forma que acontece quando você toca em um lado de uma gelatina todo o resto balança.

Acessando os Nodes do arquivo XML

Para melhorar o entendimento, pode-se dizer que node Raiz é o topo do arquivo XML, no exemplo o node Raiz é a tag <curriculo>. O código da **Listagem 4** é responsável pelo carregamento do arquivo XML. As únicas linhas que merecem atenção são as linhas 08 e 09. O método function Loadcurriculo(const FileName: string): IXMLCurriculoType; é um método global gerado pelo assistente do XML Data Binding e é o responsável por fornecer uma instância de IXMLCurriculoType, o node raiz ou base para o acesso ao nosso XML.

Além deste método, também seria possível utilizar o método function Getcurriculo(Doc: IXMLDocument): IXMLCurriculoType;, também gerado pelo assistente do XML Data Binding, sendo que este segundo normalmente é utilizado quando se recebe dados de um Web Service ou não se possui um arquivo em disco.

Já na linha 09, é feito o acesso ao valor da tag <Objetivo> do XML. A partir desse momento todas as tags existentes no XML podem ser acessadas como se fossem propriedades de uma classe comum, de modo totalmente transparente ao desenvolvedor.

Listagem 4. Abertura do arquivo XML de exemplo

```
01 procedure TForm1.btnCarregarClick(Sender: TObject);
02 begin
03   edit1.Clear ;
04   if not OpenDialog1.Execute then
05     abort;
06   edit1.Text := OpenDialog1.FileName;
07   // carrega o arquivo xml do disco
08   XMLCurriculoType := Loadcurriculo(OpenDialog1.FileName);
09   edtObjetivo.Text := XMLCurriculoType.Objetivo;
10 end;
```

Listagem 5. Recuperando valores de uma lista do XML

```
procedure TForm1.btnExperienciasClick(Sender: TObject);
var
  i: integer ;
begin
  XMLExperiencias := XMLCurriculoType.Experiencias;
  for i := 0 to XMLExperiencias.Count - 1 do
  begin
    XMLExperiencia := XMLExperiencias.Experiencia[i];
    memExperiencias.Lines.Add('Empregador:' +
                    XMLExperiencia.Empregador.NomeFantasia );
    memExperiencias.Lines.Add('Cargo:' + XMLExperiencia.Cargo.Text);
    memExperiencias.Lines.Add('Tempo no Cargo:' +
                    XMLExperiencia.Cargo.Tempo );
    memExperiencias.Lines.Add('-----------------------------------');
  end;
end;
```

Acessando listas no arquivo XML

É muito comum a existência de listas em arquivos XML, no exemplo há uma lista simples, que contém informações a respeito da experiência da pessoa. A classe responsável por esse acesso no exemplo chama-se `TXMLExperienciasType`, essa classe implementa as regras para trabalhar com listas em arquivos XML. Para acessar a lista é preciso um índice e também saber quantos itens existem nessa lista, para isso a interface disponibiliza a propriedade Count. Por padrão o primeiro índice começa sempre na posição 0.

A **Listagem 5** recupera os itens da lista do arquivo XML, também faz acesso direto às propriedades do arquivo, sem declarar explicitamente as interfaces que se deseja usar, isso economiza tempo e algumas linhas de código.

Conclusão

Com a evolução continua no desenvolvimento de sistemas e em outras áreas relacionadas com TI, podemos projetar um cenário que nos proporcionará grandes desafios. As empresas incorporarão cada vez mais sistemas e esses sistemas terão cada vez mais formatos, tamanhos e novas tecnologias embarcadas.

Aplicações legadas, adquiridas e também aplicações que serão personalizadas para atender necessidades específicas, apesar de estarem em constante evolução, em algum momento deverão interagir de modo eficiente e coerente, visando reduzir custos e principalmente, prover flexibilização ao negócio da empresa.

Dispositivos móveis, casas inteligentes, carros, enfim uma gama enorme de tecnologias faz ou farão uso desse modelo de dados para interagir e armazenar informações. Acessar dados em formato XML, de modo fácil e eficiente, permitirá ao desenvolvedor prover soluções confiáveis, elegantes, com baixa manutenção,

com menores custos. Essa é a função do XML Data Binding, prover um mecanismo eficiente e automatizado abstraindo do desenvolvedor toda a complexidade de acesso ao XML, permitindo ao desenvolvedor se preocupar com o que realmente é importante e necessário a um projeto de sucesso.

Jéfferson Ricardo Zuchi
jefferson@sygnux.com.br
Formado em processamento de dados pela FATEC-TQ. Atua como analista de sistemas Delphi na AMIL - Assistência Médica Internacional (www.amil.com.br). Também atuou como professor na Faculdade de Tecnologia de Taquaritinga(FATEC-TQ) por 2 anos, lecionando disciplinas relacionadas a Delphi e Java.

Dê seu feedback sobre esta edição!

A Clubedelphi tem que ser feita ao seu gosto. Para isso, precisamos saber o que você, leitor, acha da revista!

Dê seu voto sobre este artigo, através do link:

www.devmedia.com.br/clubedelphi/feedback

Delphi Developer's Guide to XML - Code Examples
http://keith-wood.name/DelphiXML/BookCode/index.html

XML
http://pt.wikipedia.org/wiki/XML

Delphi XML Binding Wizard Tutorial
http://edn.embarcadero.com/article/34110

W3C - World Wide Web Consortium
http://www.w3.org/

Cursos Online

webMobile PLUS

A Revista **WebMobile** oferece para seus assinantes uma série de Cursos Online de alto padrão de qualidade .

Conheça abaixo os cursos já disponíveis:
www.devmedia.com.br/curso/webmobile

Curso em .NET: **Aplicação completa de orçamento Doméstico no Visual Studio 2005**

Curso em Java: **Introdução ao uso de Web Services em Java ME**

A sua melhor opção de aprendizagem!
Assine a **WebMobile** e
Comece já seu treinamento!
www.devmedia.com.br/assine

DevMedia Mais Informações: www.devmedia.com.br/central - Tel.: 21 3382-5038

Visual Studio vs. RAD

Comparativo entre os IDEs, para desen

Ao utilizar a ajuda do Delphi Prism, nota-se que o visualizador foi remodelado para o Visual Studio 2010, a sua documentação poderá ser acessada de maneira online ou offline usando um navegador web. Além disso, permite que seja executado o download da documentação mais recente e possibilita a navegação através de um índice ou palavra-chave pesquisada. A utilização da tecla F1 está aprimorada para pesquisar o conteúdo específico de um texto selecionado.

A sua base de funcionamento está centrada no Help Library Manager (HLM) que permite pesquisar a documentação de forma a ser armazenada como conteúdo local. Ao utilizar o HLM ele gerencia a documentação local, podendo realizar atualizações através da internet ou de algum tipo de mídia, para isso, utiliza um serviço chamado BITS que realiza um acesso a internet com o objetivo de baixar o conteúdo para uso offline. Para visualizar o HLM basta utilizar o menu Help e a opção Manage Help Settings. A **Figura 1** indica as opções presentes no HLM, podendo observar se o help será executado de forma online ou offline, além da instalação de novos conteúdos e remoções.

Utilizando o Delphi XE, o menu de Help possibilita realizar consultas de ajuda, semelhante ao que ocorre no Visual Studio, a **Figura 2** demonstra o uso deste menu e suas opções de ajuda.

No menu de help ilustrado na **Figura 2** é possível verificar a pesquisa de determinadas palavras chave através do índice, uma pesquisa avançada, um gerenciador de conteúdos, ou ainda, uma pesquisa online no próprio site da Embarcadero, possibilitando assim várias maneiras de utilização do help.

Resumo DevMan

De que trata o artigo:

Este artigo indicará algumas das principais funcionalidades utilizadas para a criação de aplicações no Delphi Prism, ajudando no desenvolvimento com o ambiente do Visual Studio, onde serão destacados os pontos em comum, relacionando-os com quem possui conhecimento no IDE do Delphi XE, mas ainda não está totalmente adaptado ao ambiente do Visual Studio.

Para que serve:

Tem por objetivo diminuir o impacto visual e de ambiente ao iniciar o desenvolvimento no Visual Studio com Delphi Prism, observando o que há de diferente entre os IDEs. Dessa forma, auxiliando no que existe relacionado à criação de projetos e permitindo que se tenham condições de estar apto ao ambiente de uma forma simples e rápida.

Em que situação o tema é útil:

A principal utilidade deste artigo é diminuir a curva de aprendizado entre o desenvolvedor que utiliza o Delphi XE como seu IDE primário, mas que deseja também estender suas opções ao Delphi Prism, tendo para isso que adotar o Visual Studio. Neste caso são demonstrados os primeiros passos para o desenvolvimento com Delphi Prism baseados no conhecimento prévio em Delphi XE.

Visual Studio vs. RAD Studio:

Neste artigo serão abordados temas como o visualizador de ajuda presente no Visual Studio 2010 e que está disponível para utilização em conjunto com o Delphi Prism, da mesma forma em que no Delphi XE existe o menu de ajuda. Também é mostrado o Object Browser como facilitador nas pesquisas de objetos e membros. Além disso, ilustra-se o gerenciador de extensões, possibilitando instalar novas extensões ou funcionalidades ao ambiente de desenvolvimento do Delphi Prism, o que permite realizar um comparativo com as funcionalidades presentes em novos componentes para o Delphi XE. Serão demonstrados em ambos os IDEs, ferramentas relacionadas à arquitetura de aplicações entre outras analogias entre Delphi XE e Visual Studio.

Nota

A biblioteca HLM permite também gerenciar as configurações de visualização, possibilitando personalizar a maneira como a ajuda é apresentada ao usuário.

Nota

BITS Background Intelligent Transfer Service, é um serviço que realiza download de atualizações de conteúdo, ele verifica se um novo conteúdo online está disponível e irá atualizá-lo de forma local.

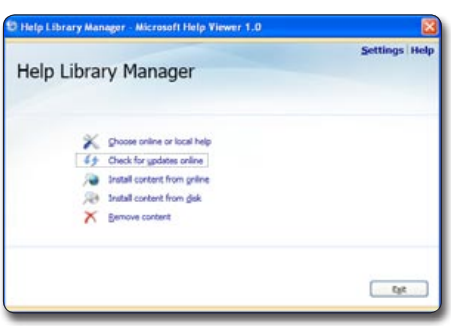


Figura 1. Opções do Help Library Manager

Studio
volvedores Delphi Prism – Parte 2

Rodrigo Araujo

Figura 2. Opções de Help no Delphi XE

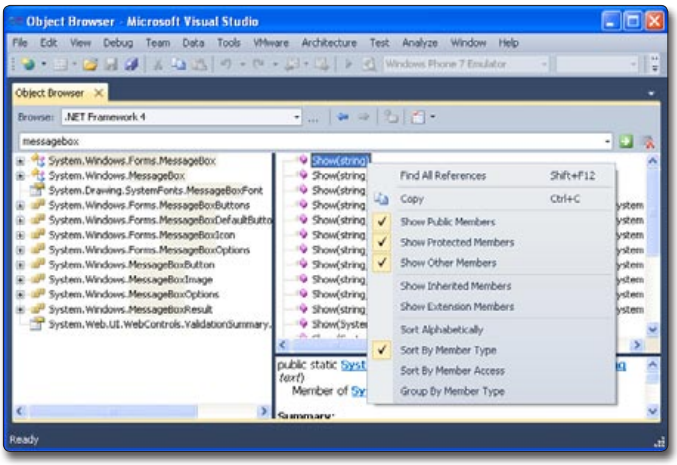


Figura 3. Object Browser, pesquisa, resultados, filtros e descrições

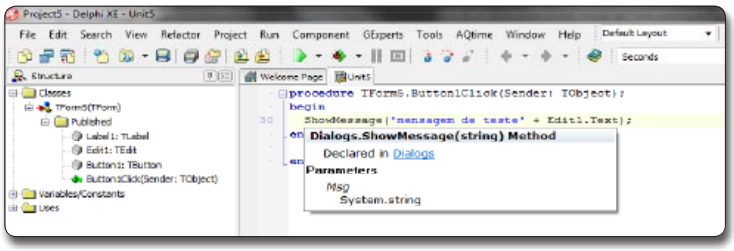


Figura 4. Visualizando informações sobre um ShowMessage

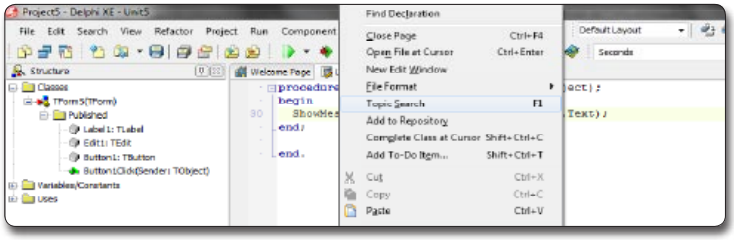


Figura 5. Opções de menu para a pesquisa da palavra ShowMessage

Object Browser

Tem por finalidade auxiliar o Visual Studio na pesquisa de palavras ou símbolos. Com ele é possível realizar pesquisas em toda a solução, por palavras utilizadas ou definidas ou então, realizar buscas de documentação. Além disso, é permitido na opção Browse indicar o formato da pesquisa, podendo ser informada a versão do .NET Framework, todos os componentes ou ainda de forma customizada (**Figura 3**).

Na **Figura 3** é possível definir uma palavra para pesquisa, onde, do lado esquerdo são retornados os resultados e ao clicar em um item, a sua descrição é apresentada do lado direito, logo abaixo, com maiores informações, caso existam. Outras configurações podem ser definidas através das opções de Object Browser, conforme a indicação do menu, definindo a visualização de tipos básicos, tipos derivados, membros públicos, protegidos e outros.

No Delphi XE, um comportamento semelhante ocorre ao utilizar uma determinada palavra, como a representada na **Figura 4** onde o uso de um ShowMessage, ao passar o mouse por cima da palavra, irá indicar algumas informações de seu uso. Esse recurso é chamado de Help Insight e existe desde o Delphi 8.

Além disso, ao clicar com o botão direito do mouse (**Figura 5**), sobre a palavra ShowMessage, um menu é apresentado com outras duas opções interessantes. A primeira é Find Declaration, que irá procurar pela declaração da palavra utilizada, e a segunda, Topic Search, indicando uma pesquisa onde os seus resultados trarão os usos da determinada palavra pesquisada.

O gerenciador de extensões

O Visual Studio 2010 possui o chamado Extension Manager, que através do gerenciador, é possível instalar extensões para o IDE do Visual Studio. Este irá realizar o download, caso seja necessário, instalar a extensão e realizar toda a manutenção nas já presentes no sistema, como desinstalar, desativar ou reativar alguma que já foi instalada, disponibilizando, assim, para o Delphi Prism. Outra opção disponível é a utilização de extensões desenvolvidas por terceiros, outros fabricantes, presentes no gerenciador através da opção de pesquisa online. A **Figura 6** acessada pelo Visual Studio 2010 através do menu Tools, indica a janela do gerenciador.

Vale notar que a janela do gerenciador é dividida em três painéis, que possuem funcionalidades distintas. O painel da direita possibilita realizar uma pesquisa por extensões, dependendo da seleção de opção no painel esquerdo, esta pesquisa será realizada por extensões instaladas, galeria online ou por atualizações disponíveis. O painel

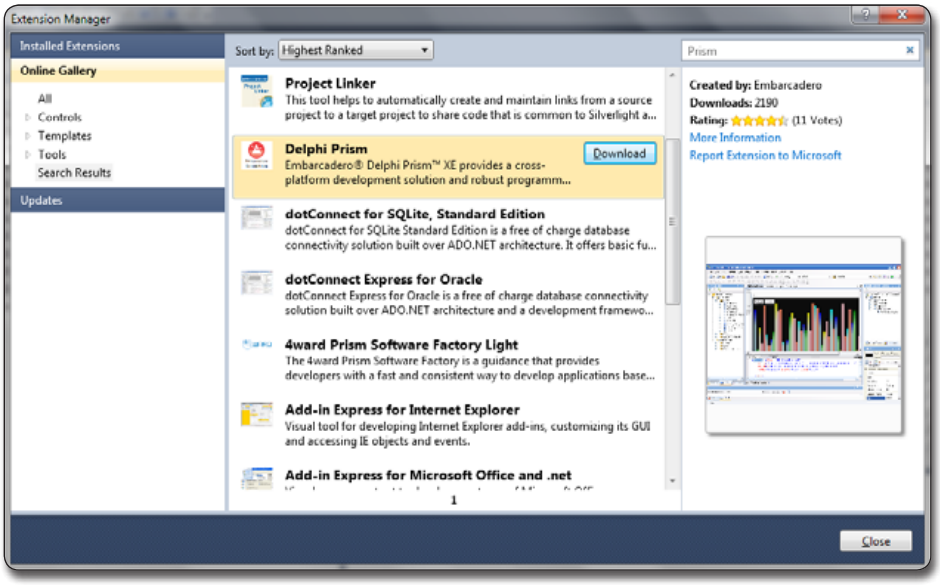


Figura 6. Gerenciador de extensões do Visual Studio 2010

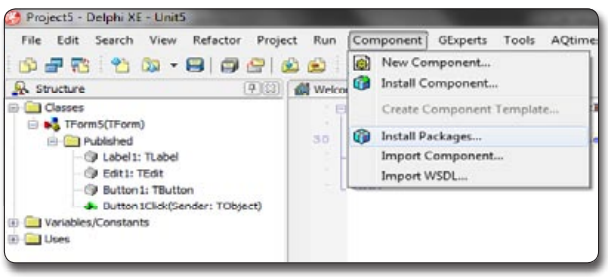


Figura 7. Uso do menu de componentes presente no Delphi XE

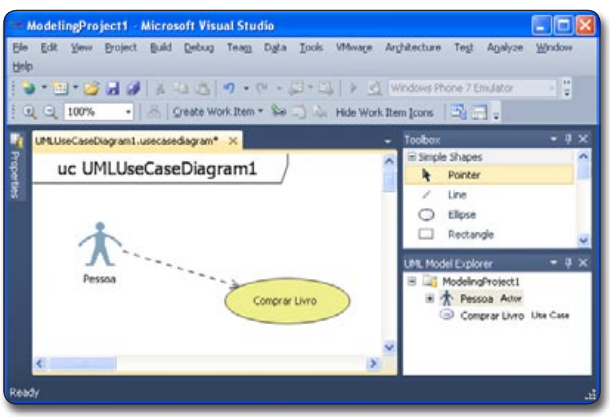


Figura 8. Definição de um diagrama de caso de uso

central exibe informações de determinados itens selecionados, de acordo com a opção escolhida no painel da esquerda, ou com os critérios de pesquisa indicados no painel da direita. Ao selecionar um item no painel central, informações sobre ele serão indicadas no painel da direita. E por último o painel da esquerda, que permite agrupamentos por extensões instaladas, extensões da galeria online e atualizações para as que já estão presentes no Visual Studio 2010.

O gerenciador de extensões suporta arquivos com extensões no formato VSIX, que é um formato de pacote que pode incluir modelos de projeto, modelos de item, item de caixa de ferramentas, componentes de gerenciamento de extensões e VSPackages. Ele também poderá carregar e instalar extensões com base no formato de instalação de arquivos MSI, mas não poderá habilitar ou desativá-los. A galeria do Visual Studio contém estes dois tipos de extensões.

De forma análoga, o Delphi XE apresenta a possibilidade de desenvolvimento de componentes e pacotes, com isso a forma utilizada no Visual Studio através de extensões fica muito próxima à ideia de componentes do Delphi XE, através do menu de componentes (**Figura 7**), é possível criar, modificar ou instalar novos componentes e pacotes.

Assim como o gerenciador de extensões apresenta a opção de instalação de extensões desenvolvidas por terceiros, no Delphi XE, isto também é possível para os componentes, e é uma opção muito utilizada, visando ampliar e aperfeiçoar o uso das ferramentas, tendo como objetivo uma melhor produtividade do que será desenvolvido.

Nota do DevMan

Um arquivo VSIX é uma unidade que poderá ser utilizada como extensão do Visual Studio, reconhecida como extensão válida e realizando a instalação do arquivo. Sua extensão é .vsix e é semelhante a um arquivo .zip utilizado para compactar conjunto de arquivos.

VSPackages são módulos de software que possibilitam estender o ambiente do Visual Studio (IDE), fornecendo elementos de interface, serviços, projetos, editores e ferramentas de Design.

Ferramentas de arquitetura

Para auxiliar na arquitetura de projetos utilizando Delphi Prism, o Visual Studio 2010 apresenta ferramentas dispostas no menu Architecture. Neste menu é possível criar novos diagramas utilizando Templates pré-definidos, entre eles, diagramas de classe, sequência, casos de uso e diagrama de camadas.

Para melhorar a comunicação da aplicação, no Visual Studio 2010 está incluído o suporte à UML (Unified Modeling Language), com isso, é possível realizar a modelagem do sistema no início ou anteriormente ao desenvolvimento do projeto.

A **Figura 8** indica um diagrama de caso de uso composto por um ator e um caso de uso representando uma operação. A janela Toolbox possibilita utilizar alguns controles como atores, casos de uso, associações e dependências. Sua representação visual é indicada ao centro ao arrastar algum controle. Neste exemplo, o ator pessoa possui um caso de uso para compra de um livro. Além disso, com o botão direito do mouse é possível criar um work item, indicando casos de testes, tarefas a serem realizadas e descrições do que deverá ser definido no caso de uso.

Outra opção interessante é o uso de diagramas de camada para visualizar o código ou definir sua arquitetura lógica de desenvolvimento do sistema. Possibilita organizar as classes, arquivos de código,

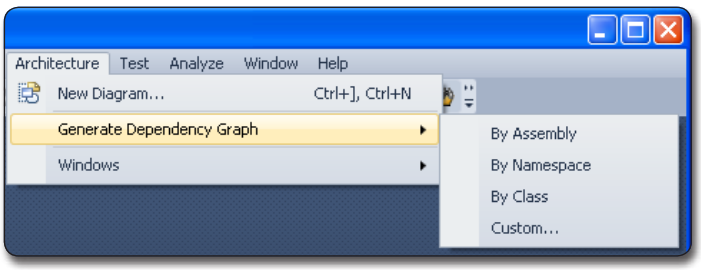


Figura 9. Menu para a geração de um gráfico de dependências

Nota do DevMan

Um work item possui as opções: bug (representa um problema no aplicativo), shared steps (indicando algo compartilhado), task (a necessidade de realizar alguma tarefa por alguém), test case (casos de teste), user story (anotações para o desenvolvimento) e issue (é uma situação que poderá interromper o desenvolvimento).

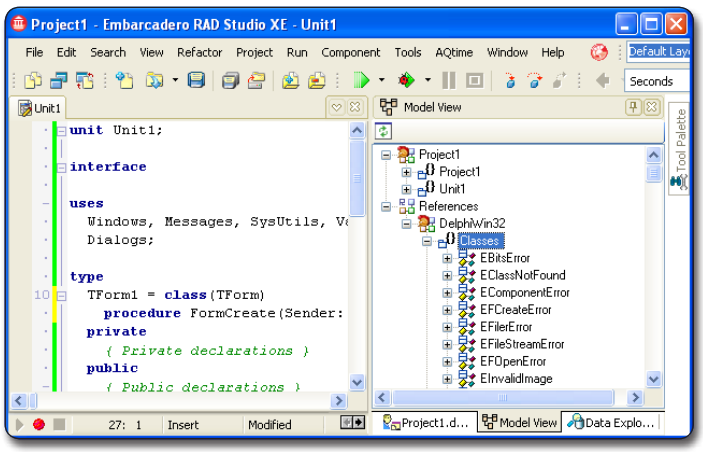


Figura 10. Model View permitindo visualizar a estrutura do projeto

projetos e outros tipos de arquivos presentes na solução em camadas lógicas e utilizar suas dependências e relacionamentos para objetivar o que será desenvolvido.

Além disso, o Visual Studio 2010 possibilita a geração do gráfico de dependências descrito na **Figura 9**, ela permite explorar de forma visual a arquitetura do projeto em questão, facilitando a sua análise e entendimento. O gráfico de dependências poderá ser gerado utilizando a opção por assembly, existindo ainda outras opções como por namespace, classes e modo customizado. Uma das vantagens na geração do gráfico de dependências é a sua forma visual de observação, onde um desenvolvedor ou analista poderá verificar o que consta e quais os relacionamentos que existem no projeto, mesmo depois dele estar desenvolvido, possibilitando entender o seu funcionamento de forma mais ágil. Outra vantagem é a constatação imediata de quais namespaces são utilizados, descritos na parte Externals. Esse diagrama é muito interessante, pois permite facilmente verificar como está o acoplamento entre classes, namespaces e assemblies do seu projeto.

Esta última funcionalidade está presente no Delphi XE através da janela de Model View onde é possível visualizar em formato de árvore a estrutura do projeto que está sendo desenvolvido, além de explorar units. A **Figura 10** demonstra a sua utilização e quais os objetos nela presentes.

É interessante visualizar a estrutura do projeto para verificar o que está presente e quais as suas funcionalidades, além do fato de que na **Figura 11** está presente o menu acionado através do botão direito do mouse, que possibilita verificar onde e como seus objetos estão sendo utilizados e tratados. É possível verificar as declarações do que está sendo utilizado, opções de copiar e colar, refactoring, propriedades e pesquisa por usos, dessa forma, auxiliando na análise do projeto desenvolvido.

A adição de referências

Para o Visual Studio carregar um Assembly através da caixa de diálogo Add Reference da Solution Explorer é bastante simples, pois o IDE, por padrão, se concentra primeiramente em carregar a guia do projeto utilizado, carregando as outras guias em segundo plano, o que diminui o tempo de leitura das outras referências. Na **Figura 12** é possível verificar que ao clicar no item de menu Add Reference, primeiramente, é aberta a aba do projeto, e em segundo plano as outras abas são carregadas.

A adição de referências irá possibilitar ao projeto que está sendo desenvolvido utilizar outras funcionalidades que não estavam presentes no projeto, normalmente utilizadas através de uma DLL que contém recursos que serão úteis.

De maneira semelhante, no Delphi XE a adição de referências ocorre ao utilizar

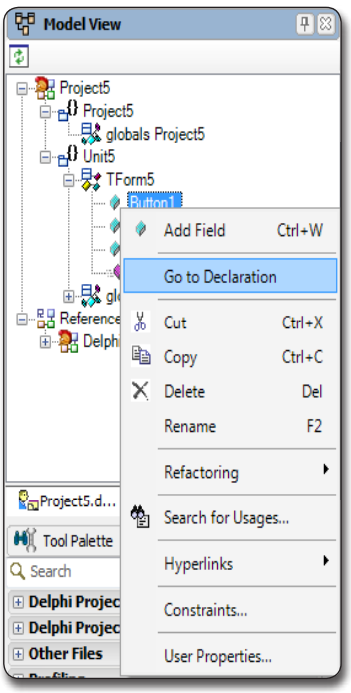


Figura 11. Menu observado ao utilizar o botão direito do mouse na janela de Model View

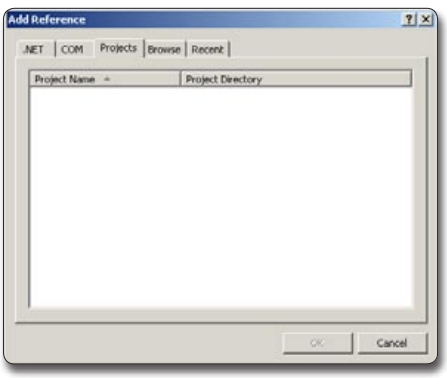


Figura 12. Adição de referências

determinados arquivos de código através da cláusula uses ou, primeiramente, através da adição de arquivos ao projeto em desenvolvimento, para em seguida estarem disponíveis para utilização. Também é possível adicionar pacotes (packages).

Depuração de código e programação paralela

Com o uso de Delphi Prism em conjunto com o .NET Framework, ocorreu o surgimento de novas bibliotecas que permitem a utilização de tarefas em paralelo em um aplicativo, ou seja a capacidade de executar múltiplas threads simultaneamente. Logo, os aplicativos poderão utilizar vários processadores sem a preocupação de trabalhar diretamente com as threads, assim, facilitando a programação com as suas bibliotecas de computação paralela. Neste aspecto, utilizando os computadores mais atuais, praticamente todos são multiprocessados, e, essa arquitetura permite a divisão de tarefas entre seus núcleos de processadores, assim, os trabalhos podem ser divididos e executados em paralelo, dessa forma, ampliando o desempenho e podendo utilizar de forma mais adequada a capacidade de processamento de um processador.

Utilizando o .NET é possível verificar a existência de uma biblioteca de classes com funcionalidades que auxiliam no desenvolvimento de aplicativos com programação paralela. A biblioteca recebeu o nome de Task Parallel Library (TPL) e possui um conjunto de tipos públicos e APIs presentes nos namespaces System.Threading e System.Threading. Tasks que simplificam o processo de adição de tarefas em paralelo. A principal preocupação de quem utilizará TPL é basicamente no desempenho do aplicativo, visto que o trabalho relacionado às Threads é gerenciado pelo próprio TPL como a divisão do trabalho, o agendamento de Threads no Pool, o cancelamento e a gestão de estado de determinada tarefa.

Algumas janelas relacionadas à depuração de código e à programação paralela estão aprimoradas. Dentre as tarefas possíveis, encontram-se, na depuração, novas tarefas que permitem visualizar janelas onde se verifica uma lista de instâncias do que está em execução em um aplicativo, a exibição das pilhas de chamadas para determinada instância de tarefas e a navegação mais fácil no código que está sendo depurado. A **Figura 13** indica duas janelas utilizadas para auxiliar na programação paralela e visualização de detalhes de forma fácil e prática.

A janela Parallel Tasks é uma janela que mostra informações de cada objeto, semelhante a uma janela de Thread, mas mais ampla, indicando tarefas que representam operações assíncronas e suas execuções de cada um desses objetos. Ela permite obter informações no momento em que o código está sendo depurado, tais informações incluem um ID, indicando o identificador da tarefa que está sendo analisada, o seu status, local de execução, uma descrição da tarefa em si e da Thread em análise e informações relacionadas ao próprio aplicativo.

Já a janela Parallel Stacks apresenta bastante utilidade para uma depuração mais visual de um aplicativo, pois nela é possível visualizar tópicos de informações das pilhas de chamadas para os segmentos do aplicativo. Permitindo navegar entre os segmentos e as pilhas de execução apresentando de forma intuitiva as tarefas e as Threads que o aplicativo possui. Em resumo, estas janelas permitem a visualização da depuração do código e o controle do que está sendo executado, além de auxiliar na observação do que está na fila para execução.

Além disso, a LINQ (Language Integrated Query), que foi uma linguagem introduzida no .NET 3.0, auxilia na consulta a diferentes fontes de dados, como bancos de dados relacionais, documentos XML ou outros tipos de objetos. A partir de sua definição, o .NET Framework 4 adicionou um novo conceito a LINQ e a programação paralela, dando origem ao chamado PLINQ (Parallel LINQ). Este novo modelo de execução de consultas é semelhante ao formato utilizado em consultas de uma expressão LINQ tradicional, a vantagem está na sua execução, que utilizará maiores recursos de processamento, como vários processadores ou núcleos de execução quando estiverem presentes e disponíveis no ambiente. A alteração na forma de realizar uma consulta é bastante pequena e o uso de PLINQ é uma ótima escolha para implementações em arquiteturas que exigem um grande processamento paralelo.

Em comparação com o Delphi Prism no Visual Studio, o Delphi XE ao realizar a depuração de código, algumas janelas semelhantes à Parallel Stacks e Parallel

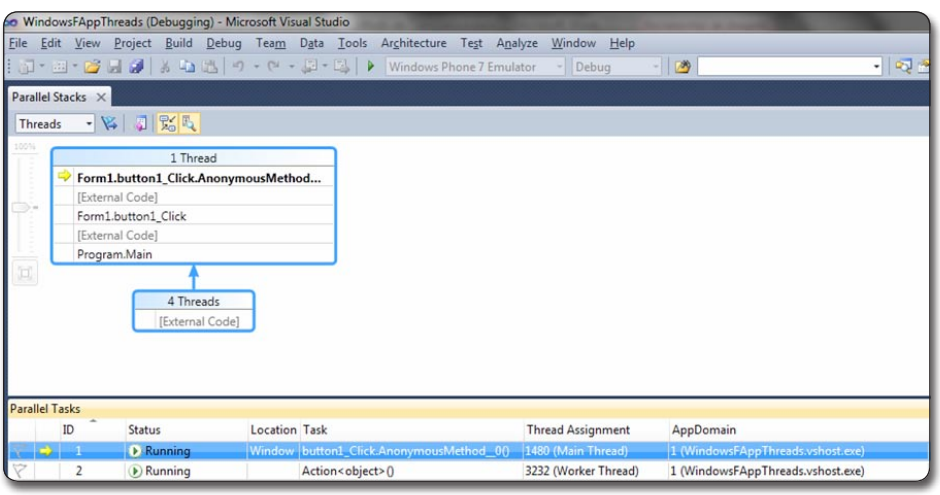


Figura 13. Apresentação da Parallel Stacks e Parallel Tasks

Tasks podem ser vistas na **Figura 14**, como a janela de Thread Status, Call Stack e Local Variables, que permitem realizar uma depuração mais refinada do que está sendo desenvolvido.

Comportamento avançado de janelas

Ao utilizar o Delphi Prism, as janelas de documentos no Visual Studio não estão limitadas ao local de edição no ambiente de desenvolvimento (IDE). Neste ambiente, principalmente as janelas de ferramentas e janelas de documentos, podem ser ajustadas para uma melhor visualização e edição de espaço para código, dependendo da forma como se deseja organizá-las. Dessa forma, as janelas podem ser arrastadas para as bordas do IDE, movidas para qualquer parte do Desktop, incluindo arrastar para um segundo monitor, facilitando a visualização do código, ao ainda com as opções lado a lado e agrupamentos, circulando livremente entre o ambiente de desenvolvimento ou sob alguma área flutuante. Obviamente, ao alterar o código em uma janela de documento, a janela de Designer também será modificada. A **Figura 15** demonstra o estilo flutuante da janela de código.

As opções de organização normalmente mais utilizadas são: janelas agrupadas para o quadro de edição, janelas de ferramentas ajustadas para a borda de um local no IDE, flutuar sobre janelas ou fora do ambiente de desenvolvimento, minimizar janelas ao longo do ambiente, exibição em monitores distintos, lado a lado e a redefinição no posicionamento para o layout padrão do Visual Studio 2010.

Ao utilizar o Delphi XE, a janela de edição de código pode facilmente ser intercalada entre o Design e o código, além da opção de visualização de histórico. A **Figura 16** demonstra a utilização básica da janela de código. Na parte inferior estão presentes as opções de Code, Design e History, permitindo de maneira simples intercalar entre elas e realizar as operações necessárias.

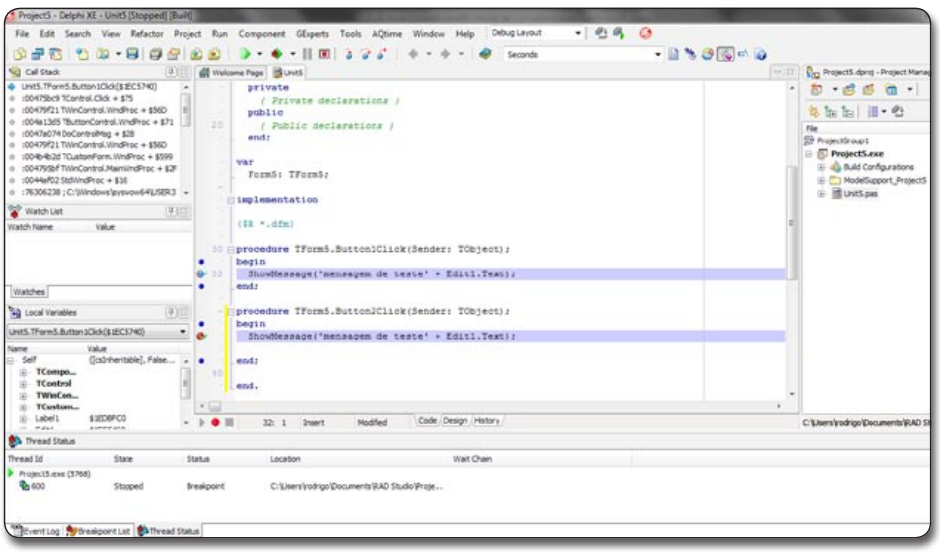


Figura 14. Apresentação das janelas relacionadas à depuração de código no Delphi XE

Conclusão

Ao utilizar o Visual Studio com Delphi Prism é possível constatar as suas funcionalidades bastante expressivas de forma visual e que irão aumentar a produtividade no desenvolvimento. Neste artigo foram abordadas algumas das principais características relacionadas ao ambiente (IDE), inclusive janelas, editor de código e menus.

Foram indicados os aperfeiçoamentos do visualizador de ajuda auxiliando nas dúvidas relacionadas com a linguagem e

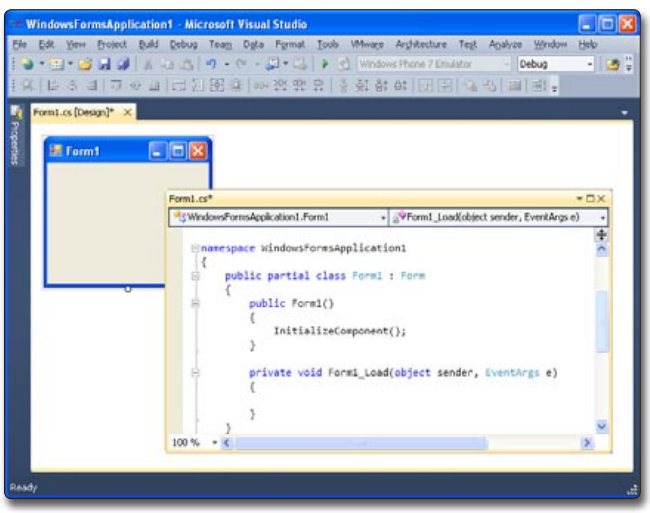


Figura 15. Janela de código em estilo flutuante podendo ser arrastada para qualquer parte da tela, inclusive outro monitor

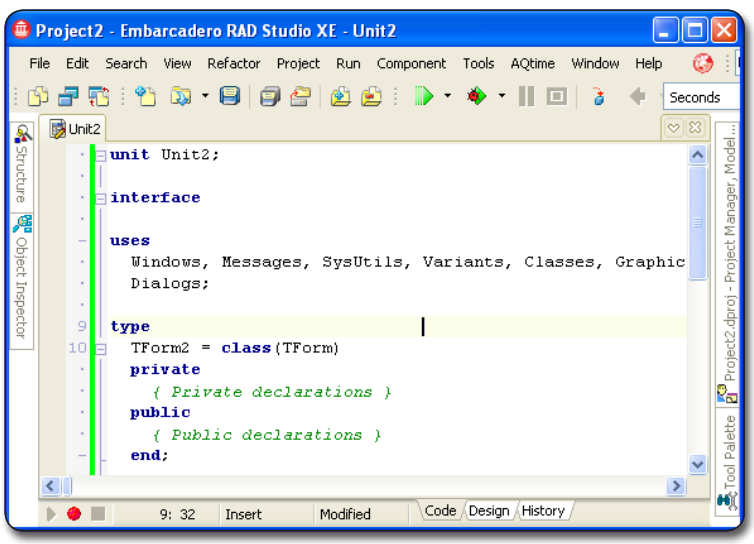


Figura 16. Janela de código no Delphi XE

ao ambiente, de forma offline ou online, comparado à ajuda existente no Delphi XE. As facilidades do Object Browser, o uso de ferramentas de arquitetura para auxiliar na análise e desenvolvimento de sistemas, o gerenciador de extensões, possibilitando utilizar novos componentes ou funcionalidades junto ao Delphi Prism semelhante ao que ocorre com os componentes no Delphi XE.

O destaque na adição de referências, possibilitando a reutilização de códigos já implementados com características distintas de uso. Um grande destaque aos principais recursos que podem ser vistos e utilizados na programação paralela, a sua utilização visando aumentar o desempenho e a depuração de um aplicativo utilizando, principalmente, as janelas Parallel Stacks e Parallel Tasks voltadas para a depuração de um sistema onde podem ser vistos maiores detalhes das operações realizadas pelos objetos de um aplicativo.

Algumas das praticidades demonstradas no artigo surgem para auxiliar, acelerar e melhorar o desenvolvimento de aplicativos. As funcionalidades descritas seguiram o padrão de comparação entre o ambiente do Visual Studio com o Delphi XE, facilitando a integração das duas culturas nestes dois ambientes distintos para a criação de sistemas. Auxiliando aos que necessitam ou querem conhecer os recursos para desenvolvimento no Delphi Prism. Com isto é possível de maneira básica, auxiliar no início da produção de sistemas com Delphi Prism, principalmente para quem já conhecia outras versões do Delphi.

Rodrigo Araujo
rodrigo.arsi@gmail.com
Bacharel em Ciência da Computação pela UFSM – Santa Maria/RS trabalha com Análise e Desenvolvimento de Software há mais de 10 anos. Desde 2003, utiliza tecnologias .NET.

Delphi Prism
http://prismwiki.codegear.com

Win32 Delphi vs. Delphi Prism
http://prismwiki.codegear.com/en/Win32_Delphi_vs._Delphi_Prism

Delphi Prism XE
http://www.embarcadero.com/products/delphi-prism

Delphi XE
http://www.embarcadero.com/products/delphi

VCL and RTL Changes for XE - RAD Studio XE
http://docwiki.embarcadero.com/RADStudio/XE/en/VCL_and_RTL_Changes_for_XE

Dê seu feedback sobre esta edição!

A Clubedelphi tem que ser feita ao seu gosto. Para isso, precisamos saber o que você, leitor, acha da revista!

Dê seu voto sobre este artigo, através do link:

www.devmedia.com.br/clubedelphi/feedback

Cursos Online

Assinatura .net magazine PLUS

Mais conteúdo .NET por muito menos!

A revista **.net Magazine** oferece para seus assinantes uma série de **Cursos Online** de alto padrão de qualidade .
Conheça abaixo os cursos já disponíveis.

Curso em destaque

Crie uma loja virtual completa

Confira neste curso Online como construir uma loja virtual completa no Visual studio 2005. Aprenda como criar o banco de dados, carrinho de compras, profile, página de erros e muito mais .

Confira o plano de aula completo:
www.devmedia.com.br/lojavirtual

A sua melhor opção de aprendizagem!
Assine a **.net Magazine** e comece já seu treinamento!
www.devmedia.com.br/assine

Outros cursos disponíveis: **www.devmedia.com.br/curso**

- Construindo relatórios com Crystal Reports e Visual Studio 2005
- Criando uma aplicação Web Completa
- Criando uma aplicação client/server no Visual Studio 2005
- Aprenda a criar um blog com ASP.NET
- Curso de C#

DevMedia Mais Informações: www.devmedia.com.br/central - Tel.: 21 3382-5038

Hospede seus sites de forma simples, rápida e ilimitada.

Plano **Básico**

- 1 domínio principal
- 50 domínios hospedados
- Transferência **ilimitada**
- Espaço **ilimitado**
- 10 contas de e-mail
- PHP, ASP e ASP.NET
- Painel de controle Plesk
- Suporte 24hs

Tudo isso por apenas

R$ **15,90** por mês
pagamento trimestral

Assine agora mesmo e descubra por que Hospedix é a hospedagem de sites preferida entre os profissionais.

- Conheça também nossos planos profissionais com registro de domínio grátis e isenção de taxas anuais enquanto for cliente.

HOSPEDIX
HOSPEDAGEM PROFISSIONAL
www.hospedix.com.br

30 DIAS GRÁTIS

Ao assinar, digite o código de desconto exclusivo para leitores desta revista e ganhe 30 dias inteiramente grátis.

RVSTMEDIA

As promoções anunciadas não são cumulativas com outras promoções e são válidas apenas para clientes Hospedix ativos.